

COVER Book Cover

VisualAge Generator
Advanced GUI Development Guide
Learning to Walk, Run, and then Fly!!!

December 1996

Document Number SG24-4238-00

NOTICES Notices

```
+--- Take Note! -----+
| Before using this information and the product it supports, be sure |
| to read the general information in Appendix E, "Special Notices"   |
| in topic E.0.                                                       |
+-----+
```

EDITION Edition Notice

First Edition (December 1996)

This edition applies to Version 2.2 of IBM VisualAge Generator Developer for OS/2. Program Number 5622-580 for use with the OS/2 Operating System.

Comments may be addressed to:

IBM Corporation, International Technical Support Organization
Dept. QXXE Building 80-E2
650 Harry Road
San Jose, California 95120-6099

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

| Copyright International Business Machines Corporation 1996. All rights reserved.

Note to U.S. Government Users -- Documentation related to restricted rights -- Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

CONTENTS Table of Contents

COVER	Book Cover
NOTICES	Notices
EDITION	Edition Notice
CONTENTS	Table of Contents
FIGURES	Figures
TABLES	Tables
FRONT_1	GUI Programming Examples and Techniques
PREFACE	Preface
PREFACE.1	How to Use This Redbook
PREFACE.2	How This Redbook Is Organized
PREFACE.2.1	Walk
PREFACE.2.2	Run
PREFACE.2.3	Fly
FRONT_2	The Team That Wrote This Redbook
FRONT_2.1	Comments Welcome
1.0	Part 1. Walk
1.1	Chapter 1. Graphical User Interfaces and VisualAge Generator
1.1.1	Characteristics of GUI Development
1.1.2	What Is a GUI Application?
1.1.3	Testing GUI Applications with ITF
1.1.4	Generating GUI Applications
1.1.5	Running Generated GUI Applications
1.1.6	Summary
1.1.7	What You Should Now Be Able to Do
1.2	Chapter 2. Introduction to the VisualAge Generator GUI Application Builder
1.2.1	Using the Parts Palette and Tool Bar
1.2.2	Parts in the Parts Palette
1.2.3	Types of Parts
1.2.3.1	Composite Parts
1.2.3.2	Data Type Parts
1.2.3.3	VisualAge Generator Member Parts
1.2.3.4	Part Settings
1.2.4	Summary
1.2.5	What You Should Now Be Able to Do
1.3	Chapter 3. Using the Standard VisualAge Generator Parts
1.3.1	Window
1.3.2	Hover Help
1.3.3	Tabbing Order
1.3.4	Parts List
1.3.5	Summary
1.3.6	What You Should Now Be Able to Do
1.4	Chapter 4. Window and Part Position Management
1.4.1	Window Types
1.4.1.1	Data Display Windows
1.4.1.2	Data Entry Windows
1.4.2	Defining Visual Parts on the Free-form Surface
1.4.2.1	Windows
1.4.2.2	Other Composite Parts
1.4.3	Defining Visual Parts inside Other Visual Parts
1.4.3.1	Fit a Part to the Parent Part
1.4.3.2	Distributing Parts Evenly on a Particular Area of a Parent Part
1.4.4	Summary
1.4.5	What You Should Now Be Able to Do
1.5	Chapter 5. Testing and Debugging GUI Applications
1.5.1	Interactive Test Facility
1.5.1.1	Using Testpoints
1.5.1.2	Using Testpoints
1.5.1.3	Working with Application Data
1.5.1.4	Controlling Test Processing
1.5.1.5	Using Trace Log Windows
1.5.1.6	Interrupting a Test Run
1.5.2	Debugging Hard Errors
1.5.2.1	Setting Up Debug Mode
1.5.2.2	Reading the WALKBACK.LOG
1.5.3	Runtime Trace
1.5.3.1	Setting up Profile Mode
1.5.3.2	Reading the TSCRIPT.LOG Profile Data
1.5.3.3	Information about GUI Load Time
1.5.4	Summary
1.5.5	What You Should Now Be Able To Do
1.6	Chapter 6. Event-Driven Programming
1.6.1	Events As the Basis of Applications
1.6.2	Connections
1.6.2.1	Creating Connections
1.6.2.2	Types of Connections
1.6.2.3	Changing Connections
1.6.2.4	Changing the Order of Connections
1.6.3	Analyzing Events in an Application
1.6.4	Designing Event-Driven Applications
1.6.5	Summary
1.6.6	What You Should Now Be Able to Do
1.7	Chapter 7. Working with Data in VisualAge Generator
1.7.1	Data Elements

1.7.1.1	Data Items
1.7.1.2	Data Types
1.7.1.3	Defining Data Items
1.7.1.4	Special Considerations for Defining Data Items
1.7.2	Working Storage Records
1.7.2.1	Accessing Data in Working Storage Record
1.7.2.2	Quick Form
1.7.3	Data Structures
1.7.3.1	Using Data Structures
1.7.3.2	Accessing Data in Data Structures
1.7.4	Occurs Items
1.7.4.1	Using Occurs Items
1.7.4.2	Accessing Data in Occurs Items
1.7.5	VisualAge Generator Tables
1.7.5.1	VisualAge Generator Table Characteristics
1.7.5.2	Using a VisualAge Generator Table
1.7.6	Ordered Collections
1.7.6.1	Characteristics
1.7.6.2	Use
1.7.6.3	Ordered Collection as a Tear-Off
1.7.7	Sharing Data
1.7.7.1	Promoting Features
1.7.7.2	Passing Data
1.7.7.3	Global Variables
1.7.8	Data Format in a GUI Application
1.7.9	Summary
1.7.10	What You Should Now Be Able to Do
1.8	Chapter 8. Procedural Logic in VisualAge Generator
1.8.1	Procedural Parts in VisualAge Generator
1.8.2	Accessing Data from Procedural Logic
1.8.3	Performing Actions Using Procedural Logic
1.8.3.1	Perform Request
1.8.3.2	Algorithms
1.8.4	Summary
1.8.5	What You Should Now Be Able to Do
1.9	Chapter 9. Visually Building Logic
1.9.1	Sequencing Actions
1.9.1.1	Order of Events
1.9.1.2	Creating Actions
1.9.2	Conditional Logic
1.9.2.1	Perform Request
1.9.2.2	Data Item
1.9.2.3	Visual Conditional Logic
1.9.3	Iterations
1.9.3.1	Iterator
1.9.3.2	Other Iterations
1.9.4	Summary
1.9.5	What You Should Now Be Able to Do
2.0	Part 2. Run
2.1	Chapter 10. Advanced GUI Features
2.1.1	Container Details
2.1.1.1	Creating a Container Details
2.1.1.2	Features of a Container Details
2.1.1.3	Scrolling Data
2.1.1.4	Editing in a Container Details
2.1.2	Drag and Drop
2.1.3	File Accessor Part
2.1.4	Closing a Window
2.1.5	Summary
2.1.6	What You Should Now Be Able To Do
2.2	Chapter 11. Building VisualAge Generator Parts
2.2.1	Embeddable Parts
2.2.1.1	Creating Embeddable Parts
2.2.1.2	Adding Parts to the Palette
2.2.1.3	Part-to-Part Communication
2.2.1.4	Sharing Data
2.2.2	Dynamic Programming
2.2.2.1	Object Factory
2.2.2.2	Difference between Open and Create
2.2.2.3	Putting Parts on a Window
2.2.3	Summary
2.2.4	What You Should Now Be Able To Do
2.3	Chapter 12. GUI Error Handling
2.3.1	Middleware and Server Database Errors
2.3.1.1	Middleware Errors
2.3.1.2	Server Database Errors
2.3.2	Data Validation Errors
2.3.2.1	Masking
2.3.2.2	Validation
2.3.2.3	Formatting
2.3.3	GUI Errors
2.3.4	Presenting Error Messages
2.3.4.1	Using Messages
2.3.4.2	Message Window

2.3.4.3	Logging Errors
2.3.5	Summary
2.3.6	What You Should Now Be Able to Do
2.4	Chapter 13. Performance
2.4.1	GUI Load Time
2.4.1.1	Loading GUI Applications
2.4.1.2	Preloading GUI Applications
2.4.1.3	Effect of Parts
2.4.1.4	Using User Think Time
2.4.1.5	Dynamic Programming
2.4.2	GUI Size
2.4.3	Logic
2.4.3.1	Procedural Logic Entry Points
2.4.3.2	Connections between GUI Applications
2.4.3.3	Unidirectional Connections
2.4.3.4	Eliminate Visual Connections for Control Flow
2.4.3.5	Use Special Actions to Populate Table and List Parts
2.4.3.6	Perform Request Considerations
2.4.4	Client/Server
2.4.4.1	Design Considerations
2.4.4.2	Data Considerations
2.4.5	Generation Options
2.4.5.1	Use the NONUMOVFL Generation Option Whenever Possible
2.4.6	Tuning Runtime Performance
2.4.7	Summary
2.4.8	What You Should Now Be Able To Do
3.0	Part 3. Fly
3.1	Chapter 14. Application Architecture
3.1.1	Building from Parts
3.1.2	Business Object
3.1.2.1	Building the Business Object Part
3.1.2.2	Business Object Part Checklist
3.1.3	Representation
3.1.3.1	Building the Representation Part
3.1.3.2	Representation Part Checklist
3.1.4	Controls
3.1.4.1	Building the Controls Part
3.1.4.2	Controls Part Checklist
3.1.5	Common
3.1.5.1	Building the Common Part
3.1.5.2	Common Part Checklist
3.1.6	Interaction
3.1.6.1	Combining the Parts in a GUI Application
3.1.6.2	Interaction Checklist
3.1.7	Application Development Considerations
3.1.7.1	Design
3.1.7.2	Documentation
3.1.8	Summary
3.1.9	What You Should Now Be Able to Do
3.2	Chapter 15. Objects
3.2.1	What Is an Object?
3.2.1.1	Real World
3.2.1.2	Abstraction
3.2.1.3	Encapsulation
3.2.1.4	Modularity
3.2.1.5	Hierarchy
3.2.1.6	Delegation
3.2.1.7	Persistence
3.2.2	Object Interaction
3.2.3	Object Terminology
3.2.3.1	The Word Object
3.2.3.2	Actions, Attributes, and Events
3.2.3.3	Class Tree
3.2.3.4	Private and Public
3.2.4	Designing for Objects
3.2.5	Summary
3.2.6	What You Should Now Be Able to Do
3.3	Chapter 16. Objects in VisualAge Generator
3.3.1	VisualAge Generator Architecture
3.3.1.1	Parts Are Classes
3.3.1.2	Tear-off Indicates Aggregation
3.3.2	Building Your Own Objects
3.3.2.1	Real World
3.3.2.2	Abstraction
3.3.2.3	Encapsulation
3.3.2.4	Aggregation
3.3.2.5	Inheritance
3.3.2.6	Communication
3.3.3	Opportunities
3.3.3.1	Using the VisualAge Class Tree
3.3.3.2	Sharing Instances
3.3.3.3	Application Objects
3.3.4	Summary
3.3.5	What You Should Now Be Able to Do

A.0	Appendix A. VisualAge Generator User Interface Standards
A.1	VisualAge Generator Limitations
A.2	National and Multiple Language Support
B.0	Appendix B. VisualAge Generator Naming Convention
B.1	Considerations
B.1.1	Requirements
B.1.2	Uniqueness
B.1.3	Meaningfulness
B.1.4	Existing Standards
B.1.5	Member List
B.1.6	Development Paradigm
B.1.7	Platform Independence
B.1.8	Aliases
B.1.9	CICS Transaction ID
B.2	Member Names
B.2.1	Descriptions
B.2.2	Records and Tables
B.2.3	GUI Names
C.0	Appendix C. VisualAge Generator Tooling
C.1	External Source Format
C.2	Print File
C.3	Summary
C.4	What You Should Now Be Able to Do
D.0	Appendix D. Attributes
D.1	Self
D.2	Object
D.3	String
D.4	Items
D.5	Menu
E.0	Appendix E. Special Notices
F.0	Appendix F. Related Publications
F.1	International Technical Support Organization Publications
F.2	Redbooks on CD-ROMs
F.3	Other Publications
BACK_1	How To Get ITSO Redbooks
BACK_1.1	How IBM Employees Can Get ITSO Redbooks
BACK_1.2	How Customers Can Get ITSO Redbooks
BACK_1.3	IBM Redbook Order Form
INDEX	Index

FIGURES Figures

1. GUI Application Definition Window 1.1.2
2. Settings Window 1.1.2
3. Pop-up Menu 1.1.2
4. The Tool Bar of the GUI Application Definition Window 1.2.1
5. Hover Help 1.3.2
6. Tab Tags 1.3.3
7. The Parts List Window 1.3.4
8. Data Display Window 1.4.1.1
9. Data Entry Window 1.4.1.2
10. Window with List Box and Push Buttons 1.4.3
11. Setting Position of the Top Edge of a Notebook 1.4.3.1
12. Sample Window with Text and Label Parts to Be Distributed 1.4.3.2
13. Form Covering the Text and Label Parts 1.4.3.2
14. Distributing the Text and Label Parts 1.4.3.2
15. ITF Monitor 1.5.1.1
16. Setting Breakpoints and Watchpoints for a Record 1.5.1.2
17. Viewing Data Items in a Working Storage Record 1.5.1.3
18. General Preferences for Testing Applications 1.5.1.4
19. Trace Entry Filter Window 1.5.1.5
20. Walkback Error Dialog for Index Out of Range Error 1.5.2.2
21. WALKBACK.LOG File for Index Out of Range Error 1.5.2.2
22. Walkback Error Dialog for Missing Public Interface Feature Error 1.5.2.2
23. WALKBACK.LOG File for Missing Public Interface Feature Error 1.5.2.2
24. A Simple Event-Driven Application (Stage 1) 1.6.2.1
25. Connect Window of the Window Part 1.6.2.2
26. A simple event-driven application (stage 2) 1.6.2.2.5
27. Settings Window of a Connection 1.6.2.3.2
28. Changing the Order of Connections 1.6.2.4
29. The Data Item Definition Window 1.7.1.3
30. Record Definition Window 1.7.2
31. Record Item Usage Window 1.7.2
32. Record Definition Window with Three Data Items Defined 1.7.2
33. GUI Application That Moves Data between a Window and a Working Storage Record Record 1.7.2.1
34. Visual Parts Created with Quick Form 1.7.2.2
35. Visual Parts Created from a Structured Working Storage Record Using Tear-Off and Quick Form 1.7.3.2
36. Working Storage Record Representing Various Forms of Aggregate Data 1.7.4.1
37. Using the Ordered Collection 1.7.6.2
38. Window for Promoting Features to the Interface 1.7.7.1
39. Window for Making a Connection to a Variable Part 1.7.7.2.2
40. The Input Field Specifications of an Entry Field 1.7.8
41. Defining a Procedural Statement by Using a Statement Template Window 1.8.3.2
42. Structure of an Event-Driven Application 1.9
43. Hierarchy of Data Items and the Effects of a Data Item Change 1.9.1.1.1
44. Event and Its Handler 1.9.1.2.1
45. Traversal of the Event Tree 1.9.1.2.2
46. Sequences of Events 1.9.1.2.3
47. Trace of Connection Triggering for GUI Programming Example 1.9.1.2.4
48. Application Structure With Embeddable Parts 2.2.1.1
49. Window for Promoting Features to the Interface 2.2.1.3
50. Window for Making a Connection to a Variable Part 2.2.1.4.2
51. Cascaded Windows 2.2.2.3.2
52. Credit Card Edit Mask 2.3.2.1
53. Three-Tier Architecture of Client/Server Applications 2.4.4.1.1
54. Application Architecture Overview 3.1.1
55. An Empty Business Object 3.1.2.1
56. Internal Structure Section of an ESF File for a GUI Application C.1
57. Method to Create Connections from an ESF File for a GUI Application C.1
58. Visual Layout Section of an ESF File for a GUI Application C.1
59. Public Interface Section of an ESF File for a GUI Application C.1

TABLES Tables

1. VisualAge Generator GUI Application Parts and Their Use 1.2.2
2. Common Attribute Settings for General Tab of Settings Window 1.2.3.4
3. Common Attribute Settings for Layout Tab of Settings Window 1.2.3.4
4. Common Attribute Settings for Drag/Drop Tab of Settings Window 1.2.3.4
5. Common Attribute Settings for Help Tab of Settings Window 1.2.3.4
6. Window Part Settings and Attributes 1.3.1
7. Window Part Attributes for Child Placement Rules 1.4.2.1
8. Trace Entry Filters 1.5.1.5
9. Data Item Specifications 1.7.1.3
10. VisualAge Generator Data Items Used in SQL Records 1.7.1.4.2
11. Data Type Relationship between GUI Application Parts and VisualAge Generator Data Items 1.7.8
12. Relative Load and Opening Times of Parts 2.4.1.3
13. VisualAge Generator Member Names B.2
14. Abbreviations of Process Options B.2.1
15. GUI Part Naming Guidelines B.2.3.1

GUI Programming Example	
Defining a GUI Application	
Saving a GUI Application	
Displaying the Settings of a Part	
Defining an Embedded GUI Application	
Defining an External GUI Application	
Testing an External GUI Application	
Generating a GUI Application	
Running a GUI Application	
Graphic Push Button with Hover Help	
Controlling Visual Part Layout and Sizing	
A Simple Connection	
Connections	
Event Trace	
Defining a Global Data Item	
Creating a Working Storage Record in the MSL	
Defining a Working Storage Record	
Moving Data between a GUI and a Working Storage Record	
Using Quick Form	
Defining a Structured Working Storage Record	
Tear-Off Attributes	
Defining Occurs Items	
Defining Multidimensional Array	
Using Occurs Items	
Defining a VisualAge Generator Table	
Using a VisualAge Generator Table	
Using Ordered Collections	
Tearing Off Ordered Collections	
Tear Off Ordered Collections	
Promoting a Feature	
Pass by Value	
Pass by Reference	
Changing the Color of a Window, Using Procedural Logic	
Understanding the Order of Connection Triggering	
Using Entry Field Objects as Events	
Ordering Connections Using a Result Attribute	
Using Procedural Code and a Perform Request Action to Open a Message Window	
Using a Data Item Toggle to Open a Message Window	
Creating a Conditional Visual Logic Flag Part	
Using the Iterator	

Seamless Scrolling	
+-----	+-----
Implementing Drag and Drop	
+-----	+-----
Using the File Accessor Part	
+-----	+-----
Using a Message Box As a Confirmation Dialog	
+-----	+-----
Using a Message Box As a Confirmation Dialog	
+-----	+-----
Defining an Embeddable Part	
+-----	+-----
Adding Function to an Embeddable Part	
+-----	+-----
Sharing Data between GUI Applications Using Working Storage Records	
+-----	+-----
Sharing Data between GUI Applications, Using Variable Parts	
+-----	+-----
Using the Object Factory to Open Windows	
+-----	+-----
Implementing Close Window Support with an Object Factory	
+-----	+-----
Using destroyPart to Manage Data Buffers	
+-----	+-----
Building Dynamic Menus	
+-----	+-----
Modifying Part Placement in a Window Dynamically	
+-----	+-----
Validation Using a Mask	
+-----	+-----
Changing a Converter	
+-----	+-----
Using a Form Input Checker	
+-----	+-----
Validating Input with VisualAge Generator Tables	
+-----	+-----
Creating a Reusable Business Data Entry Field	
+-----	+-----
Validation using Procedural Logic	
+-----	+-----
Displaying Customized Messages	
+-----	+-----
Writing a Log File	
+-----	+-----
Using Data Triggers and Connections: Part 1	
+-----	+-----
Using Data Triggers and Connections: Part 2	
+-----	+-----
Using Data Triggers and Connections: Part 3	
+-----	+-----
Sharing Data with a VisualAge Generator Table	
+-----	+-----
Dealing with Megadata	
+-----	+-----
Implementing Lookup Tables at Runtime	
+-----	+-----
Creating a Business Object	
+-----	+-----
Defining Views for a Business Object	
+-----	+-----
Creating a Controls Part	
+-----	+-----
Putting the Parts Together	
+-----	+-----
Understanding Aggregation	
+-----	+-----
Implementing Inheritance for an Ordered Collection	
+-----	+-----
Defining a Virtual Class	
+-----	+-----
Adding VisualAge for Smalltalk Classes to a VisualAge Generator GUI Application	
+-----	+-----
Editing External Source Format to Change a GUI Application	
+-----	+-----
Implementing a Simple Entry Point Application	
+-----	+-----

PREFACE Preface

This book covers the GUI application development capabilities of VisualAge Generator. It teaches you the basics and advanced techniques of the GUI application development environment and explains how to optimize the object-oriented capabilities of VisualAge Generator. The book focuses on techniques and methods that will enable you to write maintainable and reusable code, design well-performing systems, and implement objects by using GUI applications.

Many GUI programming examples are presented to demonstrate important VisualAge Generator GUI application development techniques.

This book is written for system architects, application system designers and developers, and those interested in object programming with visual programming technologies such as VisualAge Generator.

Some knowledge of VisualAge Generator Developer is assumed.

Subtopics

PREFACE.1 How to Use This Redbook

PREFACE.2 How This Redbook Is Organized

PREFACE.1 How to Use This Redbook

This book is intended to complement the *VisualAge Generator GUI User's Guide and Reference* and the help facility provided as part of VisualAge Generator Developer.

We assume that you are familiar with VisualAge Generator Developer, the use and management of members and member specification libraries (MSLs), and the process of developing and testing applications--not just graphical user interface (GUI) applications--in an OS/2 environment with VisualAge Generator.

We hope that you have reviewed the *VisualAge Generator GUI User's Guide and Reference*, especially these chapters:

- ☐ Chapter 1, "Introduction to GUI Application Definition"
- ☐ Chapter 2, "Visual Construction Fundamentals"
- ☐ Chapter 3, "Additional Techniques"

Many of the concepts discussed in Chapter 3 of the *VisualAge Generator GUI User's Guide and Reference* are also discussed in this book. Although we have tried to add value, there may be some repetition for the sake of completeness.

Most chapters in this book include GUI programming examples to help you understand and practice the VisualAge Generator GUI application programming concepts discussed. By the time you have finished reading the book, you will be a much better VisualAge Generator GUI application programmer and have an appreciation for how powerful the tool can be if you use the right approach.

We recommend that you perform all of the GUI programming examples for each chapter in sequence even if you consider yourself a skilled VisualAge Generator GUI application programmer. You will be surprised at how much there is to learn about VisualAge Generator GUI application development.

Note: Some examples begin with the code developed in previous examples. The answers are on the diskette included with this publication.

PREFACE.2 How This Redbook Is Organized

This redbook contains 379 pages. It is organized as follows:

Subtopics

PREFACE.2.1 Walk

PREFACE.2.2 Run

PREFACE.2.3 Fly

PREFACE.2.1 Walk

The first part of the book teaches you how to walk so that you have a good solid foundation before you begin more adventurous programming.

□ Chapter 1, "Graphical User Interfaces and VisualAge Generator"

Introduces the concept of GUIs and how they are developed, tested, and generated with VisualAge Generator.

□ Chapter 2, "Introduction to the VisualAge Generator GUI Application Builder"

Provides a general overview of the VisualAge Generator tool used to define GUI applications. The tool and the visual and nonvisual parts used to construct GUI applications are reviewed. Common settings for the parts used in GUI application development are listed.

□ Chapter 3, "Using the Standard VisualAge Generator Parts"

Reviews the basic visual parts used in GUI application development. GUI programming examples for selected parts are included.

□ Chapter 4, "Window and Part Position Management"

Discusses, in detail, the use of windows in a GUI application. Window control, positioning, and the relationship between a window and a GUI application are discussed.

□ Chapter 5, "Testing and Debugging GUI Applications"

Introduces the VisualAge Generator Interactive Test Facility, the tool used to functionally test, using the source code specification, VisualAge Generator GUI and non-GUI applications. The functions and features of the Interactive Test Facility are reviewed.

□ Chapter 6, "Event-Driven Programming"

Discusses the construction of GUI applications using the event-driven programming model. GUI programming examples are used to teach basic GUI application programming techniques.

□ Chapter 7, "Working with Data in VisualAge Generator"

Reviews the creation and use of data definitions in VisualAge Generator. Both basic data item and working storage record definitions as well as data manipulation in a GUI application are covered.

□ Chapter 8, "Procedural Logic in VisualAge Generator"

Discusses the use and role of procedural logic in a GUI application. Examples of using procedural logic to manipulate a GUI application are provided.

□ Chapter 9, "Visually Building Logic"

Discusses the use and role of visual logic in a GUI application. Examples of using visual logic to manipulate a GUI application are provided.

PREFACE.2.2 Run

The second part of the book teaches you how to run so that you can quickly implement functional, effective, and capable GUI application systems.

□ Chapter 10, "Advanced GUI Features"

Examines some of the more powerful parts that can be used in a GUI application. Techniques for implementing support for common programming tasks are reviewed.

□ Chapter 11, "Building VisualAge Generator Parts"

Reviews the approach used to build reusable parts, where multiple GUI applications are combined to implement a required function, with support for communication and coordination of data and actions.

□ Chapter 12, "GUI Error Handling"

Discusses the options and approaches available for implementing validation and error management in a GUI application system.

□ Chapter 13, "Performance"

Reviews the relationship between GUI application programming techniques and overall system performance. GUI application execution environment concepts and the methods of programming that can be used to improve performance are discussed in detail.

PREFACE.2.3 Fly

The third part of the book teaches you how to fly so that you can soar to new heights and implement reusable components and full systems using an object-oriented approach to VisualAge Generator GUI application development.

□ Chapter 14, "Application Architecture"

Introduces the role of a standardized approach to GUI application development, the importance of a system architecture, and how small reusable GUI applications can be effectively combined to implement the desired function.

□ Chapter 15, "Objects"

Introduces the object-oriented concepts that are a factor in building robust application systems.

□ Chapter 16, "Objects in VisualAge Generator"

Reviews the role of object-oriented programming techniques in a VisualAge Generator GUI application development environment.

FRONT_2 The Team That Wrote This Redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization San Jose Center.

Patrick F. McCarthy is a Consulting Application Development Specialist at the International Technical Support Organization, San Jose Center. He writes extensively and teaches IBM classes worldwide on all areas of application development with a specific emphasis on VisualAge Generator and related technologies. Before joining the ITSO in 1990, Pat worked in an IBM internal information systems organization (Corporate Component Procurement I/S) in Poughkeepsie, New York as a programmer, database administrator, and development center leader. Pat received a B.S. in Business Administration from the Indiana University of Pennsylvania and an M.S. in Computer Science from Marist College.

Patrick van Ginneken is an IT Specialist in the Services organization of IBM in the Netherlands. He has worked on application development projects using both VisualAge Generator and other object-oriented programming environments. For the last year Patrick has assisted customers in implementing VisualAge Generator in their organizations. He is one of the authors of an educational course on advanced VisualAge Generator GUI development. Patrick has a degree in Business Administration at the University of Twente.

Walter Janik is a Project Manager Systems Engineer in the Services organization of IBM Austria. He has worked as a Systems Engineer at IBM for 26 years, the last 10 years mainly as a project manager of system migration and application development projects. In addition Walter supported several installations as a Cross System Product (CSP) specialist and is co-author of three redbooks on CSP environments. Walter holds an M.S. degree in Mathematics and Physics from the University of Vienna.

Before we go on we must note that this book would not have been possible without the attention and assistance provided by **Alex Akilov** a member of the VisualAge Generator development team and "King-GUI" of the VisualGN CForum.

Thanks to the following people for their invaluable contributions to this project:

□ From the VisualAge Generator development lab

Tim Wilson

Henry Koch

Hayden Lindsey

Jon Shavor (for his contribution of "Tuning Runtime Performance" in topic 2.4.6)

□ VisualAge Generator fans from IBM in Europe

Erik Vos - The Netherlands

Werner Huysegoms - Belgium

John Ormerod - United Kingdom

□ Informal reviewers from inside and outside

Phong Lam

Bernie Clark

Thanks also to Maggie Cutler and Christine Dorr for their editing and manuscript support.

Subtopics

FRONT_2.1 Comments Welcome

FRONT_2.1 Comments Welcome

We want our redbooks to be as helpful as possible. Should you have any comments about this or other redbooks, please send us a note at the following address:

redbook@vnet.ibm.com

Your comments are important to us!

1.0 Part 1. Walk

This part of the book provides an introduction to GUI design and to developing GUI applications with VisualAge Generator. It describes the basic concepts and constructs of GUIs and their implementation in VisualAge Generator.

Subtopics

- 1.1 Chapter 1. Graphical User Interfaces and VisualAge Generator
- 1.2 Chapter 2. Introduction to the VisualAge Generator GUI Application Builder
- 1.3 Chapter 3. Using the Standard VisualAge Generator Parts
- 1.4 Chapter 4. Window and Part Position Management
- 1.5 Chapter 5. Testing and Debugging GUI Applications
- 1.6 Chapter 6. Event-Driven Programming
- 1.7 Chapter 7. Working with Data in VisualAge Generator
- 1.8 Chapter 8. Procedural Logic in VisualAge Generator
- 1.9 Chapter 9. Visually Building Logic

1.1 Chapter 1. Graphical User Interfaces and VisualAge Generator

Businesses are in the midst of a transition from writing host-centered business applications to writing client/server applications in distributed environments. VisualAge Generator bridges the gap between these two paradigms by using the same programming language regardless of the target platform.

Applications that traditionally ran on terminals now run on workstations. Instead of providing a text user interface (TUI), the workstation environment provides the capability for applications to use a graphical user interface (GUI) to communicate with the user. VisualAge Generator contains a powerful visual programming tool for defining both GUI applications that run on the workstation and server applications that run on workstation and host platforms.

In this book we discuss how VisualAge Generator can be used to build GUI applications. In this chapter we first discuss the characteristics of GUI development. Then we describe the components of a GUI and their characteristics and implementation in VisualAge Generator. We assume that you have a basic understanding of the VisualAge Generator Developer and know how to define member specification libraries (MSLs).

Subtopics

- 1.1.1 Characteristics of GUI Development
- 1.1.2 What Is a GUI Application?
- 1.1.3 Testing GUI Applications with ITF
- 1.1.4 Generating GUI Applications
- 1.1.5 Running Generated GUI Applications
- 1.1.6 Summary
- 1.1.7 What You Should Now Be Able to Do

1.1.1.1 Characteristics of GUI Development

Two aspects of how a user interacts with a GUI distinguish it from a TUI:

- At any time during their interaction with the computer, users are in control and decide what they want to do next. The applications are driven by events caused by user actions.
- Users use graphical elements, such as icons, pull-down menus, and radio buttons and pointer devices, such as a mouse, to communicate with the computer.

During the development of GUI applications with VisualAge Generator, components of the application to be developed are represented as graphical elements on the screen. They are created by using predefined reusable parts that are picked from a palette and dropped on that part of the screen that represents the application. Predefined components can also be used in this process. The components are connected by lines representing the type of interaction between the components.

1.1.2 What Is a GUI Application?

In VisualAge Generator, a component developed by using the VisualAge Generator GUI application definition window is called a *GUI application*. The component is stored as a member of type GUI in an MSL. A business application consists of one or more GUI applications. The GUI application definition window is used to visually develop a GUI application (see Figure 1).

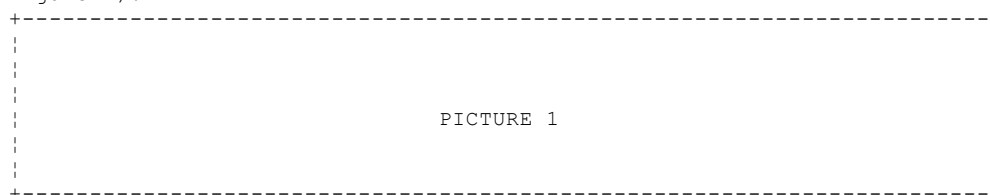


Figure 1. GUI Application Definition Window

A business GUI application communicates with the user through one or more *windows*. This is the only way in which the graphical interface is established between the application and the user. All visual parts must be materialized ultimately through a window to become visible and accessible by the user of a GUI application.

The elements that make up a GUI application are called *parts*. All parts that can be used as elements of a GUI (that is, they can be made visible and accessible to the user) are called *visual parts* or *widgets*. (1) By default the first visual part that is defined within a GUI application becomes its *primary part*. However, you can make any other visual part the primary part of the GUI application.

The default primary part when you create a new GUI application is a window. The white area surrounding the window on the GUI application definition window is called the *free-form surface*. It represents the GUI application itself, the one you are developing at the moment. On the free-form surface you define the nonvisual parts and those visual parts that are not visible to the user but are used for programming purposes. A single GUI application member can contain multiple windows. Because of the complexity this causes, we recommend that you not define more than one window per GUI application. In fact, you may find that you prefer to define more than one GUI application per window so that you can construct your application from reusable and maintainable parts. See Chapter 4, "Window and Part Position Management" in topic 1.4 for a detailed discussion on window design for GUI applications. In Chapter 11, "Building VisualAge Generator Parts" in topic 2.2 we discuss a *construction from parts* approach to GUI application development.

| GUI PROGRAMMING EXAMPLE: DEFINING A GUI APPLICATION

	<p>To define a new GUI application, do the following:</p> <ol style="list-style-type: none"> 1. Start VisualAge Generator. 2. Select or define an MSL where the members of the GUI application are to be stored. 3. From the File pull-down menu select New member... 4. In the New Member pop-up window click on the GUI radio button and the Open... push button. Then double-click on the GUI icon to open the GUI application definition window directly. <p>The GUI application definition window opens and shows a GUI application with a window as its primary part.</p> <p>The window you see is identical to that shown in Figure 1.</p>
--	---

GUI applications can be stored as VisualAge Generator members in an MSL. You have to save a GUI application before you can test, reuse, or generate it.

To run or test a GUI application, it must have a window as its primary part. The visual parts that should be visible to the user should be placed inside the window. The window is shown on the screen when you test or the user starts a GUI application.

GUI PROGRAMMING EXAMPLE: SAVING A GUI APPLICATION

To save the GUI application as a GUI application member in an MSL, do the following:

1. From the **File** pull-down menu select **Save**.
2. Because you have not named the GUI application yet, a Save As window pops up where the member name **GUIST1**. VisualAge Generator stores the GUI application in the MSL application member with the name you entered.
3. Close the GUI application definition window.

Each part has a *public interface*. The contents of the public interface of a part depend on the part's type and describe how the developer and the user can interact with the part. The public interface consists of actions, attributes, and events, collectively called *features*. Interactions between parts are visually defined by drawing connections between features of the parts. We describe the features of parts and how to draw connections in Chapter 6, "Event-Driven Programming" in topic 1.6.

Most parts, including connections and the free-form surface, have settings. The settings window represents the features of a part that can be accessed and modified during the development of a GUI application. To access the settings of a part that is not a GUI application, double-click on it with mouse button 1. If the part does not have settings, a message indicates that the part does not have any settings. Many of the settings of a part can be changed dynamically at runtime.

GUI PROGRAMMING EXAMPLE: DISPLAYING THE SETTINGS OF A PART

The Settings notebook consists of a General tab and optionally additional tabs, depending on the part's type. All tabs themselves consist of one or more pages of attribute definitions. Figure 2 shows the first page of the General tab of the Settings notebook of a window part.

To display the settings of a part, do the following:

1. From the **File** pull-down menu select **Open member....**
2. In the Open Member pop-up window enter the member name **GUIST1** and click on the **OK** button.

The GUI application with the window part is displayed.

3. Double-click with mouse button 1 on the part for which you want to display the settings window, in this case the window part.

The settings window of the window part is displayed (see Figure 2).

4. Look through all the pages of the Settings notebook and try to understand their meanings.

It is not necessary that you understand all the details of the settings window now. This exercise should give you only a feeling of the contents of the settings of a window part. If you want to know more about a setting, click on **Help**.

5. Click on the **Cancel** push button of the settings window.

The settings window is closed.

6. Double-click with mouse button 1 on the free-form surface.

Notice that the GUI application has different settings from the window. The `inheritsCommunicationsSession` attribute affects the implementation of client/server communications when remote server applications are called. Further detail can be found in the *VisualAge Generator GUI User's Guide and Reference*.

7. Click on the **Cancel** push button of the settings window.

PICTURE 2

Figure 2. Settings Window

All parts, including connections and the free-form surface, have a pop-up menu, also called a *context* menu or *object* menu. The pop-up menu is a list of frequently used functions you can use to manipulate the part. To access the pop-up menu of a part, click on it with mouse button 2. If the part has a settings window, the first selectable entry in the pop-up menu of the part is **Open Settings**. If you click on the entry with mouse button 1, the settings window is opened as if you had double-clicked with mouse button 1 on the part itself. To close the pop-up menu, click outside the pop-up menu with mouse button 1 or press the Esc key. Figure 3 shows the pop-up menu of a window part.

PICTURE 3

Figure 3. Pop-up Menu

The pop-up menu also tells you whether or not a visual part that is not contained in another part is the primary part. If it is not, the pop-up menu contains a **Become Primary Part** entry. This option enables you to change the primary part of the GUI application.

Because a GUI application is a part itself, it can be included within another GUI application. In other words, GUI applications can be nested. Nesting of GUI applications is a convenient and recommended method of reusing parts and reducing the complexity of a single GUI application. Nesting of GUI applications is described in more detail in Chapter 11, "Building VisualAge Generator Parts" in topic 2.2.

A GUI application that does not have a window as its primary part is called an *embeddable GUI application* or *embeddable part*. If an embeddable part is included in another GUI application, it is called an *embedded GUI application*. GUI applications that are embedded in a window are visible to the user.

GUI PROGRAMMING EXAMPLE: DEFINING AN EMBEDDED GUI APPLICATION

To define an embedded GUI application and use it in another GUI application, do the following:

1. From the **File** pull-down menu select **New member...**
2. In the New Member pop-up window click on the **GUI** radio button and the **Open...** push button.
3. Delete the window part from the free-form surface by selecting it with mouse button 1 (the corners of the window part will become highlighted) and pressing the Delete key.
4. Add a push button part to the empty free-form surface of the GUI application definition window by clicking on the push button icon in the parts palette and then on the free-form surface.

The parts palette is on the left side of the GUI application definition--you will see two columns of icons. They represent the categories of parts and the individual parts for use in VisualAge Generator. A push button part should be visible at the top of the right column (it is the first part in the first category). The parts palette is described in detail in "Using the Parts Palette and Tool Bar" in topic 1.2.1.

5. Save this GUI application as member name **GUIEMB1** and close the GUI application definition window.

window.

You have now created a reusable embedded GUI application. To use this embedded GUI application in another GUI application, do the following:

1. Open GUI application member **GUITST1**.
2. From the **Options** pull-down menu select **Add GUI application....**
3. Select the **GUIEMB1** member from the drop-down list in the Add GUI application window.
4. Click on the **OK** push button. Your cursor symbol becomes a crosshair.
5. Move the crosshair to the window part in **GUITST1** and click mouse button 1. The embedded GUI application part is defined at the spot of the crosshair.
6. Save the **GUITST1** GUI application and close the GUI application definition window.

HINT

Did you know that you can have a window part in an embedded GUI application? The window part can be on the free-form surface of the embedded GUI application, it just cannot be defined as the primary part. The primary part of the embedded GUI application could be a push button that when clicked opens the window. By embedding this push button in other windows, you can include in the same embedded GUI application the window opened with the push button.

If you embed a GUI application that has a window as its primary part it is called an *external GUI application*. An external GUI application cannot be placed in a container, such as a window, of another GUI application; it has to be placed to the free-form surface of that GUI application.

GUI PROGRAMMING EXAMPLE: DEFINING AN EXTERNAL GUI APPLICATION

To define a GUI application that includes an external GUI application, do the following:

1. From the **File** pull-down menu select **New member....**
 2. In the New Member pop-up window click on the **GUI** radio button and the **Open...** push button.
 3. Open the settings of the window part.
 4. Change the title of the window to "Window2."
 5. Click on the **OK** push button to save the changed title.
- To make things fun we will add the **GUIEMB1** embedded GUI application to this GUI application as well.
6. From the **Options** pull-down menu select **Add GUI application....**
 7. Select the **GUIEMB1** member from the drop-down list in the Add GUI application window.
 8. Click on the **OK** push button. Your cursor symbol becomes a crosshair.
 9. Move the crosshair to the window part and click mouse button 1. The embedded GUI application part is defined at the spot of the crosshair.

.

.

.

.
. .
.

Now to add the external GUI application...

10. From the **Options** pull-down menu select **Add GUI application....**

11. Select the **GUIST1** member from the drop-down list in the Add GUI application window.

12. Click on the **OK** push button. Your cursor symbol becomes a crosshair when over the surface. (An invalid option cursor is shown when the pointer is above the window.)

13. Move the crosshair to the free-form surface and click mouse button 1. An external GUI application part is defined at the spot of the crosshair.

14. Save the GUI application as **GUIST2** (we test the GUI applications in "Testing GUI Applications with ITF" in topic 1.1.3).

If you double-click with mouse button 1 on an embedded or external GUI application, the GUI application definition window for the part is shown. The Object menu can be used to access the settings of an embedded or external GUI application.

HINT

To change the title of a part directly without accessing its settings, keep the ALT key pressed and click on the part with mouse button 1. You can now change the title by typing in the new title you want to use.

To save the new title text, click on any spot outside the part.

(1) Although menus can be made visible to the user, we do not consider them visual parts.

1.1.3 Testing GUI Applications with ITF

One of the powerful aspects of VisualAge Generator is the fact that you can test the source code of your application independent of the runtime environment by using the Interactive Test Facility (ITF). In an ITF environment, both GUI and server applications are run in a fully interpreted mode; that is, testing using ITF simulates the runtime environment. The **goal** of the ITF is to provide a *perfect* simulation of the actual runtime environment. This goal is met in most situations. Differences can occur when there are inconsistencies in the client/server configuration, LUW management control techniques, and environment variable settings. If you think ITF is not simulating the runtime behavior of your application, use your product support process to tell the lab, which wants to know about and, if possible, correct the inconsistency.

GUI applications, just like any application written with VisualAge Generator, can be directly tested without any preparation or compilation. The immediate support for testing partial or complete applications is one of the most powerful features of VisualAge Generator. GUI applications can only be tested if they have a window as their primary part.

If you test an application, a window called the *Test Monitor* is started in the background. This window is the interface to the Interactive Test Facility (ITF) of VisualAge Generator. It allows you to track and debug your applications without generating them into executable code.

The ITF supports both the interpreted execution of GUI and server applications. Tracing of code is supported for both application types.

Of course, because the application is interpreted during testing in ITF, it is slower than during runtime. The VisualAge Generator *linkage table* can be used to make it possible for applications that are tested in ITF to directly call generated server applications in the server runtime environment. Search on the text string "Calling external programs" in the VisualAge Generator Developer help facility for more guidance on using linkage tables with the ITF.

To use a certain database during the test session when no explicit connects have been coded in the application, indicate the database name in the VisualAge Generator profile. Select **Profile** and then **Database preferences...** to change the name of the database used. The first time the database is accessed through ITF, VisualAge Generator binds the **eze2db2.bnd** package to the database.

In addition to allowing ITF to interact with the database, the bind also determines the format in which date and time values are returned to VisualAge Generator. If the format is incorrect, you can manually bind the package to the database using the correct format parameter:

```
db2 bind eze2db2.bnd datetime xxx
```

where XXX indicates the datetime format used. See the DB2 documentation appropriate for your DB2 database system for more information about binding a package to the database. The ITF is discussed in more detail in Chapter 5, "Testing and Debugging GUI Applications" in topic 1.5.

GUI PROGRAMMING EXAMPLE:	TESTING AN EXTERNAL GUI APPLICATION
	<p>To test a GUI application, do the following:</p> <ol style="list-style-type: none">1. Make sure the GUI application GUITST2 is opened.2. From the Tools pull-down menu select Test. Which window can you see? Why can't window of the external GUI application?3. Close the window (Window2) that is being tested.4. In GUITST2 double-click with mouse button 1 on the external GUI application GUITST2 GUI application definition window opens for the GUITST1 GUI application.5. Test GUITST1. Which window can you see now?

See "Running Generated GUI Applications" in topic 1.1.5 for information on calling generated applications from the IDE.

1.1.4 Generating GUI Applications

Generating a GUI application creates Smalltalk code that can be loaded into (bound to) the runtime Smalltalk image for VisualAge Generator GUI applications. The generated files have an extension of .app and include all the parts used in the GUI application that are not embeddable parts, VisualAge Generator tables, or non-GUI applications.

Embeddable parts reside in their own separate .app file. They get generated automatically if they are embedded in a GUI application that you generate and you set the *Embedded GUIs* check box in the **Generated Code** tab in the generation options notebook. This equals the /GENEMBEDDEDGUI generation option. VisualAge Generator tables and non-GUI applications are generated separately. The generated VisualAge Generator table has an extension of .tab. Non-GUI application generation output is dependent on the target runtime environment. See *Generating VisualAge Generator Applications* for details.

The different generated parts (GUI applications, VisualAge Generator tables, and applications) are independent. If you change one embeddable part, you do not have to regenerate the GUI applications that embedded the reusable part--unless of course they have also changed.

If the GUI application calls a server application, the called applications should be defined in the linkage table used during generation. You can identify the linkage table file to be used as input on the *Input Files* tab in the generation options notebook. This equals the /LINKAGE=filename generation option. See *Developing VisualAge Generator Client/Server Applications* for more information about using linkage files with GUI applications.

Any GUI application that has a window as the primary part causes a command file to be generated when the target environment is OS/2. The command file can be used to start up the application. The command file contains only the following line:

```
@eze2run open APPNAME
```

If you generate a GUI application that uses an object factory to create parts at runtime, ensure that you generate these dynamically created embeddable parts or GUI applications as well.

| GUI PROGRAMMING EXAMPLE: GENERATING A GUI APPLICATION

To generate a GUI application, do the following:

1. Close any open GUI application definition window and the Test Monitor window.
Build a member list of the GUI applications you have defined so far.
2. From the **File** pull-down menu select **Member list...** to open the Member List Criteria dialog.
3. Define the Member name as ***** and GUI applications as the only selected Member Type by selecting the **none** radio button and then the GUI toggle button. Click on the **list...** push button. A member list window will be shown.
4. Select the **GUIST1** GUI application from the member list.
5. From the **File** pull-down menu select **Generate...** to open the Generate dialog for GUI application **GUIST1**.
6. Prepare the generate request:
 - a. Select a Target system of OS/2 and provide appropriate file names for the Command Message log files.
 - b. Do not select any of these toggle buttons: **Append to command file, build command only**, or **LAN based generation**.
7. Click on the **Set options...** push button to open the OS/2 Generation Options notebook.
8. Select (if not already selected) the **Embedded GUIs** toggle button on the **Generated Code** notebook page.
9. Define a directory name in the **Generation output directory** entry field on the **Output** notebook page.
10. Click on the **OK** push button to close the OS/2 Generation Options notebook.

11. Click on the **OK** push button of the Generate dialog to start the generation command. Answer **Yes** to the confirmation dialog. This will start the VisualAge Generation Monitor which will generate the **GUITST1** GUI application.
12. If you selected the **View messages** toggle button on the Generate dialog, you will see the View Messages dialog with the messages received during generation. Click on the **OK** push button to close the View Messages dialog.
13. Click on the **Cancel** push button to close the Generate dialog for GUI application.
14. Close the VisualAge Generation Monitor by clicking on the **Stop monitor** push button. Answer **Yes** to the confirmation dialog.

1.1.5 Running Generated GUI Applications

To run a generated GUI application in an OS/2 environment, execute the corresponding command file. This action will start the VisualAge Generator runtime image (stored in the EZE2RUN file) and load the necessary GUI application .app-files (which the runtime image therefore must be able to locate). If your application calls server applications the linkage file indicating the location of the server applications should also be accessible. The active linkage table at runtime is the first found of:

1. The linkage table identified during generation, minus the path information, is searched for in the current directory and then the DPATH.
2. The linkage table referenced by the active setting of the CSOLINKTBL environment variable. If path information is not provided, DPATH is searched.

The ITF can call generated server applications if the called application name is in the linkage table identified in the ITF general preferences profile. Linkage table processing for an ITF call differs from runtime.

ITF uses the existence of the application in the linkage table identified in the ITF general preferences profile to determine whether a call is to be a remote call. If the a generated application is to be called, ITF passes the linkage table file name, less the path information, to the runtime remote call support routines. These routines then process like runtime, using the linkage table name passed by ITF.

Note: This processing does not guarantee, or require, that the linkage table identified in the ITF general preferences profile be the same referenced by the CSOLINKTBL environment setting.

The VisualAge Generator runtime image (EZE2RUN) includes all the generated Smalltalk code for the base parts of VisualAge Generator. The Smalltalk image (.app) of every GUI application that is used thereafter is loaded into memory as well. This memory is not released unless you perform the destroyPart action on a window or any of its subparts. See Chapter 13, "Performance" in topic 2.4 for a more extensive discussion on memory use of VisualAge Generator GUI applications.

The database that is used during runtime execution of the application is determined by the EZERSQLDB environment variable when no explicit connect has been coded in the application. The EZERSQLDB environment variable can be changed before you start the application.

Note: Environment variable changes made in a command session (OS/2 window) are local. Global settings must be made in the CONFIG.SYS file.

Other environment variables can affect database selection for VisualAge Generator applications. Please review *Running VisualAge Generator Applications on OS/2, AIX, and Windows* for additional guidance.

```

GUI PROGRAMMING EXAMPLE:   RUNNING A GUI APPLICATION

-----

To run a GUI application, do the following:

1. Open an OS/2 window and change to the generation output directory identified during application generation.

2. Issue the dir GUI*.* command, and you should see these files, which were created during generation:

    □E:\vgenout\vgui+dir GUI*.*

The volume label in drive E is TPAD-E-DISK.
The Volume Serial Number is A6D5:B014.
Directory of E:\vgenout\vgui

12-07-96      5:51p          3330             0  guiemb1.app
12-07-96      5:51p          3886             0  guitst1.app
12-07-96      5:51p           23             0  GUITST1.cmd
        3 file(s)              7239 bytes used
                                259958784 bytes free

```


The .APP files are the Smalltalk code for external GUI application **GUITST1** and embedded GUI application **GUIEMB1**. A GUITST1.CMD file was also generated for external GUI application **GUITST1**.

3. Edit the GUITST1.cmd file to see:

```
@eze2run open GUITST1
```

The command tells VisualAge Generator to use the Smalltalk GUI application runtime image to open (load, create, and open) the generated Smalltalk code for the **GUITST1**. GUI application. The **GUIEMB1** GUI application will also be loaded since it is embedded in **GUITST1**.

4. Enter the command **GUITST1**. You should see a window with a push button, just as you tested the **GUITST1** GUI application.
5. Close the window and issue this command to force a shut down of the Smalltalk GUI application runtime image:

```
EZE2RUN KILL
```

1.1.6 Summary

Building applications with a GUI requires different programming techniques and design techniques from those required to build TUI applications. You can use VisualAge Generator to build both GUI and TUI applications. In this book we focus on building GUI applications.

An application system consists of one or more GUI applications. A GUI application is stored as a VisualAge Generator member of type GUI in an MSL. GUI applications contain parts. A part has a public interface that enables parts to interact through connections. A part's interface consists of actions, attributes, and events, which are the features of the part. Each part, including a connection, has settings.

Only visual parts defined on the free-form surface can become the primary part. A GUI application has one and only one primary part. A GUI application must contain at least one visual part. An executable GUI application must have a window as its primary part. For a part to become visible to the user it must be either a window or a visual part that is ultimately embedded inside a window.

GUI applications can also contain one or more embedded or external GUI applications. A GUI application can be included in more than one GUI application. An embeddable GUI application does not have a window as its primary part. An embeddable GUI application can be embedded in a window. An external GUI application has a window as its primary part. An external GUI application cannot be embedded in a window, but you can make it part of the GUI application by placing it on the free-form surface.

1.1.7 What You Should Now Be Able to Do

You should now be able to:

- ☐ Tell the difference between a GUI application and a window
- ☐ Describe the purpose of a GUI application
- ☐ Tell the difference between an embedded and an external GUI application
- ☐ Relate a GUI application to a VisualAge Generator MSL member
- ☐ Create a GUI application member.

1.2 Chapter 2. Introduction to the VisualAge Generator GUI Application Builder

A GUI application consists of parts and the connections between the features of these parts. As there are various requirements to the functions of a GUI application, there are different types of parts with specialized properties to fulfill these requirements. In this chapter we show you how to use the VisualAge Generator GUI application definition window (or the *GUI builder* as it is also called) to define the parts of your application and discuss their specific characteristics.

Subtopics

- 1.2.1 Using the Parts Palette and Tool Bar
- 1.2.2 Parts in the Parts Palette
- 1.2.3 Types of Parts
- 1.2.4 Summary
- 1.2.5 What You Should Now Be Able to Do

1.2.1 Using the Parts Palette and Tool Bar

The parts palette of the GUI builder window provides access to all parts that can be added to a GUI application. The parts are assigned to categories. The left side of the parts palette indicates the categories, and the right side, the parts in the selected category. Selecting another category brings up the parts of that category.

To add a part to a GUI application, select it from the parts palette and move the mouse pointer over the location where you want the part to appear. If a part is a nonvisual part, you cannot drop it onto a part that includes other parts such as a window. In such cases the pointer turns into a no-entry sign.

You can place parts on the free-form surface. Although the user will not be able to see the parts, they can be useful for programming the logic of your program. We discuss building logic into an application in more detail in Chapter 6, "Event-Driven Programming" in topic 1.6 and Chapter 9, "Visually Building Logic" in topic 1.9.

If you want to add multiple parts to the GUI application without having to select the part every time, select the **Sticky** setting underneath the parts palette. This action places a part of the type selected from the parts palette in the GUI application every time you click mouse button 1 on the free-form surface or in a part that can include other parts such as a window. Deselect the setting to stop adding multiple parts.

```
+-----+
|  HINT  |
+-----+
|        |
+-----+
|        | The Sticky setting can also be used to apply the same font or color setting to multip
+-----+
|        |
+-----+
```

You delete a part from the GUI application by selecting it and either choosing **Delete** from the Object menu or pressing the Delete key. When you delete a part, you also delete any subparts it contains, associated promote part features, and connections.

Once you have dropped the part in the GUI application, you can select it by clicking on it with mouse button 1. The selected part will be highlighted with handles in the form of filled in boxes at the corners.

You can select multiple parts by holding down the Ctrl key and clicking on other parts. In this way you build up a set of selected parts. You deselect a part by holding down the Ctrl key and selecting the part again. You deselect multiple parts by holding down mouse button 1 and dragging the pointer over the parts to be deselected. A part can be added to a set only if it is included in the same part (for example, parts in the same form) as the other selected parts.

If you have selected one or more parts, you can change their layout. You can move parts by dragging them to a different location. You can size them by dragging one of the handles (black box) to a different spot. You can align, size equally, and distribute selected parts over the surface of the part in which they are included by using the different options from the tool bar. The primary part (last part selected) in a set determines to which part the other parts are aligned or sized. The primary part in a set of selected parts will be highlighted with handles in the form of filled in boxes at the corners. The other selected parts will have handles in the form of outlined boxes. The handles of this part are filled boxes instead of outlined boxes. Figure 4 shows the different types of actions available on the tool bar.

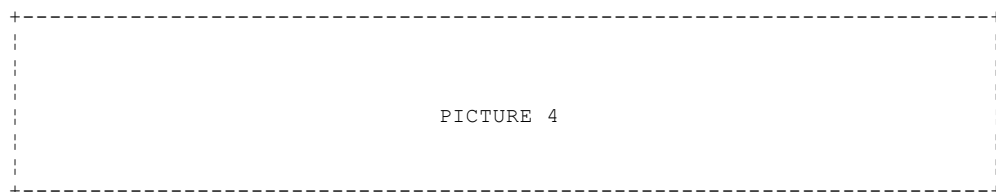


Figure 4. The Tool Bar of the GUI Application Definition Window

You can also distribute parts within the bounding box of a set of selected parts. The bounding box is defined by the top left-hand corner and bottom right-hand corner of the set of selected parts. From the Object menu of a set of selected parts, select **Layout** and then **Distribute** and then either **Horizontally in bounding box** or **Vertically in bounding box**, depending on the layout. The **Horizontally in surface** and **Vertically in surface** options are identical to using the distribute options in the tool bar.

If you make an error when adding, removing, or manipulating parts, you can undo and redo the changes using the respective options from the **Edit** menu. You can even get a list of all the actions that can be undone or redone.

1.2.2 Parts in the Parts Palette

VisualAge Generator provides a lot of different, useful parts for building a GUI. Deciding which part to use in which case is an important and nontrivial aspect of building your GUI applications. You will find the book entitled *Object-Oriented Interface Design, IBM Common User Access Guidelines* a useful guide in making such decisions. It explains the common user access (CUA) standards to which the interface should comply and when certain types of parts should be used. We strongly urge you to read that book before you start building GUI applications. In Appendix A, "VisualAge Generator User Interface Standards" in topic A.0, we list all of the limitations of VisualAge Generator in respect to the CUA standards.

The VisualAge Generator parts may not map one-to-one to the parts mentioned in *Object-Oriented Interface Design, IBM Common User Access Guidelines*, and there are nonvisual parts that are specific to VisualAge Generator that are mainly used for implementing logic. Table 1 lists all of the VisualAge Generator parts, the parts palette category to which they belong, and their use.

Table 1. VisualAge Generator GUI Application Parts and Their Use		
Part Category	Part Type	Use
Button	Push Button	Enables a user to start an action. If a push button is on a window or a menu, the same action should be available through the menu. The push button can have text, a graphic, or both to indicate its function.
	Toggle Button	Enables a user to make a choice between two states. The toggle button is used when there is a clear, distinguishable choice. Toggle buttons are placed in a group.
	Radio Button	Enables a user to indicate a choice between mutually exclusive textual choices.
	Scale	Enables a user to change a quantity in relation to a range of quantities.
	Slider	Enables a user to change a quantity in relation to a range of quantities.
	Hot Spot	Enables a user to start an action. Use a hot spot to designate an area as having the ability to accept user interaction. Often used along with icons or graphical images in the user interface.
Data Entry	Text	A field in which the user can enter text.
	Formatted Text	A field in which the user can only enter text that complies with a predefined format.
	Multi-line Edit	A field in which the user can enter multiple lines of text that can be edited.
	Table	A grid that can be used to display an array of information. The container details part provides similar a display with improved function.
	Label	Static text that can be used to indicate the meaning of another part.
	Spin Button	Enables a user to scroll through a list of values within a range. Use in situations where the range is limited and the increments are logical.
List	List	Can contain a list of data with one column from which the user can select one item.
	Multiple Select List	Can contain a list of data with one column from which the user can select multiple items.
	Drop-down List	Can contain a list of data with one column from which the user can select one item. Only the selected item is visible in the display area. The list contents is dropped down when requested by the user. The user can then select an item from the list. This selected item is then visible in the display area when there is no space for a regular list.
	Combo Box	Can contain a list of data with one column from which the user can select one item. Only the selected item is visible in the entry area. The list is dropped down when requested by the user. The user can then select an item from the list. This selected item is then visible in the entry area. The user can also choose to edit the contents of the entry area manually. The user does not have to match an item in the list contained in the combo box. Use in situations where there is no space for a regular list.
	Spin Button	Enables a user to scroll through a list of values within a range. Use in situations where the range is limited and the increments are logical.
	Container Details	A grid that can be used to display an array of information. Columns are displayed separately and are as such separate parts. Use to represent a list of data.

VisualAge Generator GUI Development Guide

Parts in the Parts Palette

		that can be edited or in which the columns must be resizable.
	Container Details Column	A column in the container details part
Menu	MenuBar	This part is a menubar that can be dropped on a window to attach it directly to the window or dropped on the free-form surface. When dropped on a window, a menu is shown in the window, and menu items can be added directly to it. When it is dropped on the free-form surface, it must be associated with the window separately through a connection.
	MenuBar Item	An item in the MenuBar that the user can choose. These items are routed to other parts that lead to another menu.
	Popup Menu	A menu that can be associated with a menubar item or cascade menu item. It can be cascaded or pop-up menu. In itself this part is not visible. Menu choices, menu cascade, menu toggle, or separator parts can be added to it.
	Menu Choice	An action choice in a menu. Can be part of a pop-up menu or a MenuBar item.
	Menu Cascade	A routing choice in a menu leading to another pop-up menu.
	Menu Toggle	A setting in a menu that can be either checked or cleared. Avoid using this part. Instead, include settings as toggle buttons in a settings window.
	Separator	A separator line in a menu distinguishing different parts of the menu. When used in a Window (instead of a menu), the separator can also be used as horizontal and vertical lines, dotted lines, and double lines.
Canvas Category	Window	The part that is ultimately seen by the user. A window can include other parts.
	Form	A part that can include other parts. It is useful for containing visual elements that have no formal relationship.
	Group Box	A part that can include other parts. It is useful for containing visual elements that have a formal relationship. The relationship can be named in the Group Box.
	Scrolled Window	A part that can include other parts. When the window in which this part is placed is sized smaller than the size of the part, it shows scroll bars to allow the user to use to reach an area of the scrolled window if it becomes hidden. There is a setting that will make the scroll bars only appear as needed.
	PM Notebook	An OS/2 style notebook that can include multiple pages of information, including the settings pages of a part. This part is valid in OS/2 and all Windows environments.
	Windows Notebook	A Windows style notebook. This part is valid in OS/2 and all Windows environments.
	Notebook Page	A page in either the PM or Windows Notebook
Prompters	Message Prompter	A message box with a number of predetermined possibilities for icons, text, and push button sets.
	Text Prompter	A message box that requests a line of information from the user and returns it to the GUI application from which it was initiated.
	File Selection Prompter	This part represents the standard OS/2 or Windows (depending on your environment) file dialog. It enables a user to select a file to be used. It does not support reading from and writing to the file itself.
Models	Ordered Collection	An unsorted but indexed array that can contain any kind of information.
	Object Factory	A part for creating parts at runtime
	Variable	A part that can point to any other part and so act as a representation of the part to which it points
	File Accessor	A part that allows you to read and write information from and to a file. If file specifications are not provided, it brings up the standard operating system file dialog.
	Container	An-OS/2 specific part that can be used to represent data or objects in the system as icons. The user can interact with the icons. Multiple icons can be viewed in a hierarchical tree structure.

VisualAge Generator GUI Development Guide

Parts in the Parts Palette

OS/2	OS/2- Windows Notebook	A notebook that automatically has one page added to it. The page is r independent part. This is the old VisualAge Generator implementation notebook and should be used only when you understand the implications: will run faster than the PM or Windows Notebook parts but does not sup Windows 95 or Windows NT platforms.
Data Member Parts	Record Member Part	A representation of a VisualAge Generator record member such as a work record. Occurrences within different GUI applications are independent other.
	Table Member Part	A representation of a VisualAge Generator table member. Each occurren VisualAge Generator table with the same name uses the same data. They independent by GUI application.
Logic Member Parts	GUI Application	A GUI application can be embedded in another GUI application. The GUI application can be either external or embeddable.
	Callable Function	Represents a callable server application. The callable application ca written in any language, including VisualAge Generator.
	Process Member Part	Includes procedural logic using the procedural language of VisualAge G
	Statement Group Member Part	Includes procedural logic using the procedural language of VisualAge G
External Functions	DDE Client	Dynamic data exchange (DDE) is a protocol for exchanging data between applications that run on your workstation. This part allows your Visu Generator application to communicate with DDE server applications.
	DDE Server	DDE is a protocol for exchanging data between applications that run on workstation. This part allows your VisualAge Generator application to DDE server application.
Multimedia	Compact Disc Player	A representation of the compact disc player in your PC as a device. I you to interact with the compact disc player, to play its music, for e
	Video Disc Player	A representation of a video disc player connected to your PC as a devi allows you to interact with the video disc player, to play the video i it, for example.
	Audio Wave Player	A device for using audio wave files. The file contains instructions f sounds. This part allows you to manipulate, for example, play, the fi
	Digital Video Player	A device for using an digital video file. The file contains instructi showing movies. This part allows you to manipulate, for example, play
	Video Playback Window	A part that allows you to view the output of the video disc player or video player.
	Motion Buttons	A standard set of buttons like those found on your video or CD player
	Track Buttons	A standard set of buttons for advancing and going back a track, for in your compact disc player.
	Frame Buttons	A standard set of buttons for advancing and going back a frame, for in your video disc player.
	Other Multimedia Buttons	A collection of all other possible multimedia buttons, such as buttons recording, playing back, and muting.

1.2.3 Types of Parts

In this section we examine some distinctions between the parts in the parts palette.

Subtopics

1.2.3.1 Composite Parts

1.2.3.2 Data Type Parts

1.2.3.3 VisualAge Generator Member Parts

1.2.3.4 Part Settings

1.2.3.1 Composite Parts

The first important distinction is between parts that can hold other parts (composite parts) and parts that cannot. A part that allows other parts to be included within it has a relationship with those subparts: It is their parent. Deleting the parent also causes the subparts to be deleted. The following parts are composite parts:

Container details	Window	Form
Group Box	Scrolled Window	PM Notebook
Windows Notebook	Notebook Page	OS/2 Container
OS/2-Windows Notebook	GUI Application	

A composite part can include other parts. Any alignment or distribution that is performed on those other parts is performed within the composite part. Most composite parts can also be included in other composite parts; for example, a window can include a group box that includes a number of entry fields. In fact, if you want to have an embeddable part that includes multiple entry fields, you must place them in a composite part such as a form. The form will be the primary part in the embeddable part. When this part is embedded in another GUI application, all entry fields included, along with the primary part, will be visible in the embedding GUI application.

1.2.3.2 Data Type Parts

Some parts hold information that must conform to a certain data type. For example, an entry field can be set to accept only those entries that conform to a certain data type. This behavior is determined by an attribute of data type parts called a converter. The following parts have a converter:

Push Button	Toggle Button	Radio button Set
Entry Field	Formatted Text (*)	Multi-line Edit
Table Column	Label	Spin Button
Menubar Item	Menu Choice	Menu Cascade
Menu Toggle	List Box	Multiple Select List
Drop-down List	Combo Box	Container Details Column

(*): The formatted text part has a converter that differs from the converter of the other parts. Instead of checking the format after the user has provided an entry, the formatted text part converter does not allow the user to enter any character that does not conform to the definition of the data type.

1.2.3.3 VisualAge Generator Member Parts

Some parts are also VisualAge Generator members and are thus stored in an MSL. Their definition is stored separately from the GUI application in which they are used. These parts can therefore be edited by choosing them from the member list or as part of the GUI application. Some of these VisualAge Generator member parts even have to be generated separately when preparing the runtime application. The following parts are also VisualAge Generator members:

+-----		
GUI Application	Callable Function	Table Member Part
+-----		
Record Member Part	Process Member Part	Statement Group Member Part
+-----		

1.2.3.4 Part Settings

When you place a part from the parts palette on the GUI application, the GUI builder defines default values for many of the attributes of the part. You may want to change some of the default definitions according to your use of the part. You can see and change the definitions for these attributes in the settings window of the part.

If you open the settings window of a part, you get a number of tabs and pages (each tab can contain multiple pages). A number of the settings are generic for the part with which you are working. A number of characteristics always have the same meaning, however. Table 2 through Table 5 present the names, default settings, and descriptions of the most commonly used attributes in the settings windows of the different parts. For full descriptions, see the *VisualAge Generator GUI User's Guide and Reference*.

Table 2. Common Attribute Settings for General Tab of Settings Window		
Attribute	Default Setting	Description
Label type	<blank>	Indicates whether a graphic image or text string is shown in the part. If Image is selected, there are options for the image. If Text is selected, there are options for the text string.
Label string	partName	The label that is shown in the part
Data type	<none>	The data type to which any data represented in the part should conform. If the data does not conform to a string showing ** error ** is shown. You can customize the data type to indicate formats, valid ranges, and default values by clicking on the Customize... button.
Reformat data on focus change	on	Indicates whether the data should be reformatted according to the data type settings when the field loses focus
Mnemonic	<blank>	Indicates the letter that is part of the label that will get an underscore. The mnemonic can be accessed by pressing the letter when the group in which the part is included has the focus or combining the key with the Alt key when the part does not have the focus.
Accelerator key	<blank>	Indicates the key combination that enables a user to access the function quickly.
Graphic type	none	Defines the type of graphic for the part. Possible selections are Icon, Image, or None. If you select a Graphic type other than None, you have to specify the path and file name of the graphic part you want to use.
File name	disabled	If you have chosen Image as the Graphic type, define in the File name the path and the file name of the graphic part or the dynamic link library module name. If you have chosen Icon or Bitmap, the integer in the ID field identifies the resource in the DLL for the graphic part.
Shading type	disabled	Defines the shading intensity of the graphic part. Possible selections are Normal, or Dark. Normal is the default.
Initial items	<blank>	The initial items to be displayed in a list. Each item should be put on a new line.
Attribute to display	<blank>	The attribute of the part connected to the list that should be displayed
Character limit	depends on part	The maximum number of characters that can be entered in this field
Read-only	off	Indicates whether the value of this part can be changed by the user
Signal events on each keystroke	on	Signals a change in the data each time a new character is entered in the field. The change can be used to execute some logic.
Selection technique	depends on part	Indicates the way in which the user can select items from the list, for example, single, multiple, or extended
Border margins	0	Controls spacing between child and parent parts
Border shadow	depends on part type	Controls spacing for three-dimensional appearance
Label alignment	center	The alignment of the label within the part

Recompute size	off	Indicates whether a part always attempts to fit exactly around a new label.
Part name	partNamen	This is the internal name of the part as it is known within the GUI application.
Border width	0	The width of the border around the part in pixels. Can be used to add a border around a label part.
Part enabled	on	The user can interact with the part.
Traversable	depends on part type	Traversable and Tab group work together to achieve the tabbing and focusing of a part. You can use the cursor movement keys to navigate through those parts that have Traversable set. If a part has Tab group set, the cursor movement keys will move over those parts within the group of parts. This attribute can also be set from the Object menu of a part.
Tab group	depends on part type	Traversable and Tab group work together to achieve the tabbing and focusing of a part. You can use the Tab and Backtab keys to move to a part or group of parts that has Tab group set. After you tab to a group of parts, you can use the cursor movement keys to navigate through the parts that are traversable and support the use of the cursor movement keys. This attribute can also be set from the Object menu of a part.

Table 3. Common Attribute Settings for Layout Tab of Settings Window

Attribute	Setting	Description
Coordinates		Specifies the position of the part. Available when the part is not included in another part. Indicates the top left-hand corner in pixels.
Dimension		Specifies the size of the part. Available when the part is not included in another part. Indicates the width and height of the part in pixels.
Edge selection	top	Attachments allow the definition of relationships between parts to keep the layout when resizing the window or parts of the window. This setting indicates the edge for which the settings of the attachment apply. Choose another edge to change the corresponding attachments.
Attachment type	depends on edge	Attachments allow the definition of relationships between parts to keep the layout when resizing the window or parts of the window. This setting indicates whether the selected edge is connected to nothing, another edge of itself, its parent, or another part.
Offset to target	depends on current layout	Attachments allow the definition of relationships between parts to keep the layout when resizing the window or parts of the window. This setting indicates the offset between the part indicated as the attachment and this part in the layout. Either this or the proportional offset should be indicated if the part is attached.
Proportional offset	depends on the current layout	Attachments allow the definition of relationships between parts to keep the layout when resizing the window or parts of the window. This setting indicates the offset is proportional to the size of the parent. Either this or the absolute offset should be indicated if the part is attached.
Target parts	<none>	Attachments allow the definition of relationships between parts to keep the layout when resizing the window or parts of the window. This setting indicates the part to which this part is connected. This setting is only available if you have chosen to attach the part to another part.

Table 4. Common Attribute Settings for Drag/Drop Tab of Settings Window

Attribute	Settings	Description
Allow drag	off	Indicates whether dragging from this part is allowed. If dragging is allowed, you can set the valid actions: moving, copying, or linking. For a drag-and-drop action to take place the target must support one of the same actions.
Allow drop	off	Indicates whether dropping on this part as a target is allowed. If dropping is allowed, you can set the valid actions: moving, copying, or linking. The source must support one of the same actions in order for a drag-and-drop action to take place.

Table 5. Common Attribute Settings for Help Tab of Settings Window

Attribute	Settings	Description
-----------	----------	-------------

Filename	<blank>	Indicates the name of the file that will be searched for the help text requests help for this part
Topic panel ID	<blank>	The name of the panel within the file with which this part is associated panel will be shown when a user requests context help for this part.
Keys panel ID	<blank>	The name of the panel within the file with which this part is associated panel will be shown when a user requests help for keys for this part.
Title text	<blank>	The text shown in the title bar of the help window that is displayed when requests help

<u>HINT</u>		
		VisualAge Generator does not allow you to change the default values of the parts. However, you can create a GUI application that includes all of the parts that you use in your application development and give them the correct default settings.
		Put this common part GUI application in a common MSL and always have it open when you are developing an application. You can then copy and paste the parts from the common part application instead of obtaining parts from the parts palette. This will ensure that the parts have the correct initial settings.
		You could get a similar result by promoting parts to the palette. Just remember that VisualAge Generator updates often reset the palette to what was shipped with the product (so that new parts to the product).

1.2.4 Summary

A GUI application consists of parts. The part in which all parts are ultimately shown is a window. Parts are accessible through the parts palette. Parts can be added to a GUI application, deleted, moved, aligned, and distributed.

Parts have different uses. The CUA standards define the use of parts in the context of your application.

Parts can be categorized in a number of ways. A part is either a composite part or not. If it is a composite part, it can include other parts. A part either has a data type or not. If it has a data type, the type can be set and must be adhered to by the user. A part is either individually stored in the MSL or as part of the GUI application. Parts that are stored in the MSL can be edited outside the context of the GUI application.

Each part has settings. The settings determine the initial layout and behavior of the parts. Most parts have similar types of settings.

1.2.5 What You Should Now Be Able to Do

You should now be able to:

- ☐ Identify the basic tool bar control functions of the VisualAge Generator GUI application builder
- ☐ Distinguish between the different parts and understand their uses
- ☐ Understand the basic characteristics of the parts.

1.3 Chapter 3. Using the Standard VisualAge Generator Parts

A GUI application consists of parts and connections between the features of those parts. As there are various requirements to the functions of a user interface, there are different types of parts with specialized properties to fulfill these requirements. In this chapter we review the basic use of some of the parts available in the GUI builder.

We do not discuss the basic use of the GUI application definition or the parts palette. These topics are covered in Chapter 2, "Visual Construction Fundamentals," of the *VisualAge Generator GUI User's Guide and Reference*.

Subtopics

- 1.3.1 Window
- 1.3.2 Hover Help
- 1.3.3 Tabbing Order
- 1.3.4 Parts List
- 1.3.5 Summary
- 1.3.6 What You Should Now Be Able to Do

1.3.1 Window

The part that forms the base for your GUI application is the window.

A window is the only valid part for running or testing a GUI application. All user functions have to be visualized somehow by means of components of a window so that they become visible on the screen and accessible to the user. A window holds other parts. Any parts placed inside a window automatically have a relationship with the window, their parent.

Table 6 lists the default settings of the attributes of a Window part that describe the appearance and behavior of the window.

Table 6. Window Part Settings and Attributes. This table describes the attributes of a Window part that describe the appearance and behavior of the window.		
Attribute	Default Setting	Description
Minimize button	On	The window has a Minimize button.
Maximize button	On	The window has a Maximize button.
System menu	On	The window has a System menu.
Title bar	On	The window has a Title bar. The Title bar is used not only to display the title of a window but also to move the window on the screen (by clicking on the Title bar with mouse button 1 and dragging the window to a new location). A window without a Title bar cannot be moved on the screen.
Resize handles	On	The user can resize the window. The Resize handles appear if the user places the mouse pointer over the border of the window. By clicking mouse button 1 and dragging the mouse, the user can change the size of the window.
Border	Inactive	The window gets a thick border in the same color as the Title bar. If the window has no Resize handles. Therefore the size of a window without a border cannot be changed. Border and Resize handles can only be selected together alternatively. Note: In the Windows runtime environment, windows that use the Inactive setting cannot have a menu.
Start iconic	Off	At startup the window is displayed as an icon (minimized) rather than as a window.
Allow resize	Off	The window changes its size automatically when its contents change.
Title text	Window	This is the title that is displayed in the Title bar. You can specify a title if the window does not have a Title bar.
Icon Graphic type	None	Defines the type of graphic for the System menu and the icon of the window. Possible selections are Icon, Bitmap, Image, or None. If you select a Graphic type other than None, you have to enter the file name of the graphic part you want to use (see next row: File name).
Icon File name Module name, ID	Disabled	If you have chosen Image as Graphic type, define in the File name field the path and the file name of the bitmap image. If you have chosen Icon or Bitmap as Graphic type, in the Module name field define the path and name of the resource file DLL and in the ID field define the integer that identifies the resource number for the graphic part. Use the Find... push button for help in finding the graphic. If the name of a module is found, the file name without its extension is returned. If the graphic part is found, it is displayed to the user in the ID entry field.
Icon Shading type	Disabled	If you have chosen a Graphic type other than None, define here the shading intensity of the graphic part. Possible selections are Normal, or Dark; Normal is the default.
Wallpaper style	Normal	Defines the style of the window's background. Possible selections are Normal, Tiled, or Scaled.
Wallpaper Graphic type File name Module name, ID Shading type	None	Defines the graphics of the window's background. For more information about the definition of the Wallpaper attributes, see the description of the Icon attributes in this table.
Part name	Window	This is the internal name of the window as it is known within the application.

Part enabled	On	The user cannot interact with the window if it is disabled.
		state the window cannot be closed, and the parts in the window
		accessed. However, the user can size, move, maximize, and mi
		window if the window is disabled.

As for all parts, it is advisable to set standards within your organization for the settings of the parts to ensure that all windows in an application look and feel the same to the user.

The sizing and positioning of windows and other visual parts can affect the look and feel of the GUI application for different users who use different resolution settings for their displays. Designing flexible GUI applications requires careful planning and consideration. We review this topic in detail in Chapter 4, "Window and Part Position Management" in topic 1.4.

A window can be opened in different modes. This modality determines the way in which the user can or has to interact with the window. The mode VisualAge Generator uses is determined by the action that is used to open the window. Windows in VisualAge Generator support the following types of modes:

System Users cannot access any other windows that are open in the operating system until they have closed the window opened in this mode ([openSystemModalWidget](#)).

Full Application Users cannot access any other windows that are open in the current application (the VisualAge Generator GUI runtime environment) until they have closed the window opened in this mode ([openFullApplicationModalWidget](#)).

Application Users cannot access the window that opened the application modal window until they have closed the window opened in this mode ([openApplicationModalWidget](#)).

Modeless The windows mode does not restrict the user from acting on any other windows ([openWidget](#)).

Owned A window opened from another window as owned will always stay on top of the window that owns it ([openOwnedWidget](#)). The window opened as owned is modeless.

From a user interface perspective, modeless windows are preferred except in situations where you are prompting the user for information or indicating the occurrence of an error. In the latter case, use the Full Application mode.

1.3.2 Hover Help

VisualAge Generator provides hover help over parts that allow both a graphic and a textual representation, like a push button, and of which the graphic representation has been chosen. Hover help is help that appears when the user holds the mouse pointer over a part until text appears in a text box "hovering" over the part (see Figure 5).

The text that is used for hover help is the text that has been entered as the label string of the part. Although the part is shown as graphic, you can switch between these two modes and define the label string.

Hover help is shown only when the composite part that holds the part for which you want hover help has the hoverHelpEnabled attribute set to true.

GUI PROGRAMMING EXAMPLE: GRAPHIC PUSH BUTTON WITH HOVER HELP	
	<p>To give a push button hover help, do the following:</p> <ol style="list-style-type: none"> 1. Create a new GUI application. 2. Open the settings of the window part and set the check the toggle button Enable hover all children on the second page of the general tab in the settings notebook. 3. Put a push button on the window. 4. Open the settings of the push button and change the label string to "Closes the application." 5. Choose the Graphic radio button for the label type option. 6. Choose the Image radio button for the graphic type option. 7. Select a bitmap file (.BMP) using the Find... push button and then save the push settings. (There should be some bitmaps in your c:\os2\bitmap subdirectory; flames is a nice choice.) 8. Click on the Ok push button in the settings notebook. 9. Test the application. (Save the GUI application as HOVERHLP.) Hold the mouse cursor over the push button. VisualAge Generator will show you the text you entered as the label string for the push button. <p>Note: Clicking on the push button will not close the application because we did not make the required connection.</p> <p>Figure 5 shows the result of this example.</p>
	

Figure 5. Hover Help

1.3.3 Tabbing Order

In addition to using the mouse to move between parts of the window, users can use the Tab key. The sequence in which parts get the focus when the Tab key technique used is called the *tabbing order*. VisualAge Generator allows you to change the tabbing order of the parts. (2)

Each composite part has a tabbing order related to its parts. The tabbing order is initially equal to the order in which you have added the parts. You can change the tabbing order manually or let VisualAge Generator determine the default tabbing order (from left to right and from top to bottom). To let VisualAge Generator determine the tabbing order, select **Set Tabbing** followed by **Default Ordering** from the Object menu of the composite part.

To manually adjust the tabbing order, select **Set Tabbing** followed by **Show Tab Tags** from the Object menu of the primary part. Each part that has been set to be traversable gets a numbered tag indicating its order in the tabbing sequence (see Figure 6).

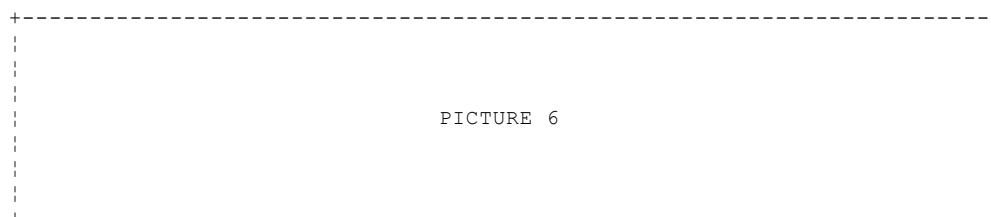


Figure 6. Tab Tags

Drag the number of a field to a different location to change the tabbing order. The target of this operation gets the indicated number in the tab sequence. You cannot drag the number outside the part in which it is included. Instead use the number of the composite part, which is represented in blue.

If a part is traversable but does not have the tab group attribute set, it is shown as part of a group. Use the tab key to access the group. Press the tab key again to leave the group. Within the group you can use the cursor movement keys to move between the parts. A group is specified with sequence numbers indicating the group and the order in the traversal sequence (for example, 1:2).

The order of traversal can be changed similarly to the way you change the tabbing order. A group always has the same tab group indicator (1) if it is not part of its own composite part. The only way you can have another part earlier in the tab sequence is by putting the group in a composite part.

(2) The tabbing order also determines the order in which parts are created.

1.3.4 Parts List

A GUI application can include quite a number of parts. The GUI builder parts list allows you to access parts without visually locating them. To bring up the parts list, select **View Parts List...** from the Object menu of a composite part. A tree with all the parts included within it is shown. The tree can be expanded or collapsed. The parts are shown in the tabbing order of the parts (see Figure 7).



Figure 7. The Parts List Window

In the parts list each part has an Object menu that enables you to open the settings of the part, change its name, or delete it. You can also change the part name by holding down the Alt key and clicking mouse button 1 just as you do to change the title of a window in the GUI application.

<u>HINT</u>	
	The parts list can be invaluable for finding parts that have been covered by other pa
	can also use the parts list to save a <i>lost</i> part, that is, a part where one of the dim
	become 0 and you no longer can see or manipulate it in the GUI builder visual view.

1.3.5 Summary

Many modes can be used to open a window. Each mode can alter the flow of how a user interacts with the GUI application system. The window part is used to show other parts to a user.

Parts that can have both graphic and textual representations can be provided with hover help. The tab sequence of the parts can be adjusted and the parts included in a GUI application can be viewed through the Parts List window.

1.3.6 What You Should Now Be Able to Do

You should now be able to:

- ☐ Distinguish among the different parts and understand their uses
- ☐ Understand the basic characteristics of parts
- ☐ Use hover help
- ☐ Control the tabbing order of your window and understand its implications
- ☐ Use the GUI builder parts list.

1.4 Chapter 4. Window and Part Position Management

In this chapter we provide recommendations for designing and implementing windows that are:

- ☐ Easy to define and maintain
- ☐ Self-adjusting to different target environments and operating systems
- ☐ Self-adjusting to different screen resolutions
- ☐ Adjustable to the individual needs and preferences of users

First we discuss the design aspects of defining windows and the visual parts within a window. Then we describe the properties of a window related to window management and how to use them effectively. Finally we suggest techniques for defining visual parts within a window so that they will adjust automatically to changes in the runtime environment.

To complement this chapter review these sections in Chapter 3, "Additional Techniques," of the *VisualAge Generator GUI User's Guide and Reference*:

- ☐ "Sizing and Positioning Visual Parts"
- ☐ "Establishing Position Where Windows Appear"
- ☐ "Sizing Embedded GUIs."

Subtopics

- 1.4.1 Window Types
- 1.4.2 Defining Visual Parts on the Free-form Surface
- 1.4.3 Defining Visual Parts inside Other Visual Parts
- 1.4.4 Summary
- 1.4.5 What You Should Now Be Able to Do

1.4.1 Window Types

If, as the starting point for window design, we consider the typical tasks of users using business applications, we can distinguish the following two types of user interactions:

- ☐ Find information, that is, searching for a set of existing data
- ☐ Manipulate data, that is, entering new data and modifying existing data.

To provide the user with a graphical interface to perform these tasks, we therefore can distinguish the following two types of windows:

- ☐ Data display windows
- ☐ Data entry windows

Subtopics

1.4.1.1 Data Display Windows

1.4.1.2 Data Entry Windows

1.4.1.1 Data Display Windows

Data display windows are used to display more than one instance of business information (for example, data from multiple rows of a relational table) in one active window. Data is presented in containers such as lists and tables. Usually the user cannot enter or change data directly in a data display window but must switch to a data entry window to modify data. A data display window is sometimes called a "List View Window."

Figure 8 shows a typical data display window.

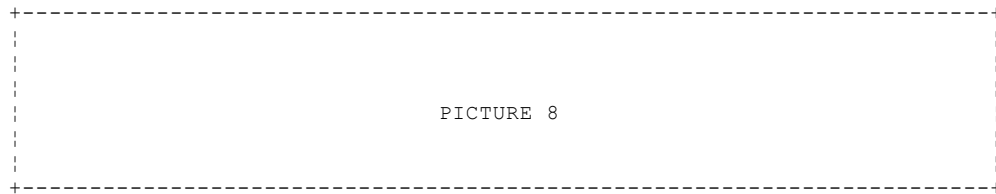


Figure 8. Data Display Window

1.4.1.2 Data Entry Windows

Data entry windows are used to display, change, or enter data related to one instance of business information (for example, data from one row of a relational table) in one active window. Data is presented in fields or with other graphical controls. A window of this type is sometimes called a "Detail View Window."

Figure 9 shows a typical data entry window.

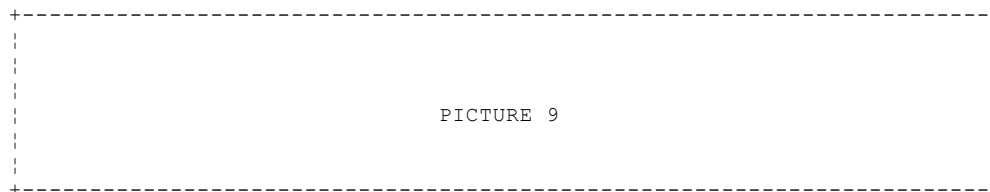


Figure 9. Data Entry Window

Our initial suggestion is that you develop data display and data entry functions as separate components that can be reused.

In addition to the data display and data entry components used to exchange data between the user and the GUI application, a window has parts that enable users to:

- ☐ Control the type of interaction they want to perform (for instance, display or modify data)
- ☐ Navigate between windows
- ☐ Adjust the interface to their individual needs and working style

These parts will be used in both data display and data entry windows.

You may decide to define windows that combine the features and functions of both data display and data entry windows. However, consider these issues before deciding to mix the window types:

- ☐ With GUI applications, users may have many windows open concurrently. They will arrange the windows to meet their needs, so that each is available when required. Therefore there is no longer a need to have all information and functions on one window as there is in a 3270 environment where users can only have one window available to them at a time. Placing the information and functions of a GUI application in many small windows rather than in a few large windows increases user flexibility.
- ☐ Overloading GUI application windows so that they contain as much information and function as possible (a typical approach for those with 3270 text user interface experience) is not necessarily a good approach. If you want to design GUI applications that can be developed and maintained effectively, start with simple and easy-to-use windows.
- ☐ Overloaded windows are difficult to define, maintain, change, understand, and use.

Decisions regarding the separation of data display and data entry windows must be incorporated as part of your local standards for GUI application development so that you can:

- ☐ Keep the appearance of GUI applications consistent for the user
- ☐ Keep the windows small and simple
- ☐ Develop small GUI parts
- ☐ Develop reusable parts
- ☐ Increase programming productivity
- ☐ Reduce maintenance effort

Our suggestion to separate data display and data entry windows is only one technique for making GUI windows as simple as possible. We also suggest that you consider using only a subset of the parts available for GUI application development. This will increase both programmer productivity and the usability of the GUI application.

In the sections that follow we describe the visual parts used in the data display and data entry windows and these development issues related to the process of building dynamic GUI applications.

1.4.2 Defining Visual Parts on the Free-form Surface

You can define any part on the free-form surface, but not all parts will actually be shown to the user. In this section we discuss the options available for controlling the position and sizing of visual parts that will be shown to the user. This includes windows in external GUI applications and other nonwindow primary parts in embedded GUI applications.

Subtopics

1.4.2.1 Windows

1.4.2.2 Other Composite Parts

1.4.2.1 Windows

The attributes of a window part that are used for positioning and sizing a window are defined in the **General** and **Layout Tabs** of the Settings window. We describe the attributes and their use in "Window" in topic 1.3.1.

When you define a window, VisualAge Generator creates default settings for its attributes. Some of the default settings do not always facilitate adaptable and flexible window management, however, so we recommend that you change some of them:

Allow resize	Change to On
Title text	Provide a unique title for your window
Part name	Provide a unique name based on your local naming standards

Some visual parts, such as a window located in the Canvas category of the parts palette, can contain other visual parts. These parts have some special attributes, called *Child Placement Rules*, in their General settings. There is no need to change any of the default settings.

Table 7. Window Part Attributes for Child Placement Rules			
Attribute	Window		
	Settings	Change	Change
Space between Adjacent Parts - Horizontal	0	No	
Space between Adjacent Parts - Vertical	0	No	
Proportional Sizing - Fraction Base	100	No	

If you want to change the Layout settings of a window, consider the following rules:

Proportional grid not selected

X and Y represent the coordinates of the upper-left corner of the window relative to the screen in pixels. Width and Height represent the size of the window in pixels.

Proportional grid selected

X and Y represent the coordinates of the upper-left corner of the window relative to the screen as a percentage of the size of the screen.

Width and Height represent the size of the window as a percentage of the size of the screen.

Notes:

1. If one of the values of coordinates X and Y is set to 0 or has been erased, the window will pop up at a random position every time it is opened.
2. If you set Width or Height to 0, the window shrinks to a size that is unusable.

A value of 50 for the Width and Height with Proportional grid displays a window with a size that is half the width and height of the screen. A value of 40 displays a window with approximately the same size as the default window in a new GUI application.

We recommend selecting Proportional grid for Coordinates and Dimensions value control. This selection allows the window to adapt automatically to varying screen sizes and resolutions. Additionally, if you want VisualAge Generator to display the window at a randomly selected position on the screen, you can erase (or set to 0) the Coordinates X and Y entry fields.

Note: First check all toggle buttons for Proportional grid and then change the contents of the entry fields. Apply your changes and look at the new window layout first before you click on OK to ensure that your changes are acceptable.

1.4.2.2 Other Composite Parts

Some visual parts, such as a window, a Form, and a Group Box of the Canvas category, that contain other visual parts have some special General settings attributes called *Child Placement Rules*. The settings suggestions for windows (accept the defaults, see "Windows" in topic 1.4.2.1) also apply to the Form and Group Box parts.

The other attributes of interest for positioning and sizing visual parts are defined in the Layout settings of the part. The content of the Layout settings of a visual part depends on the placement of the part. If the part is defined on the free-form surface, the Layout settings describe the coordinates of the upper-left corner and the size of the part. Note that these parts do not offer a Proportional grid specification in their settings.

1.4.3 Defining Visual Parts inside Other Visual Parts

All visual parts that are not windows, or embedded GUI applications, can be defined inside other visual parts. This can be a nested process. However, all visual parts other than the window itself or embedded GUI applications must ultimately be contained in a window to be accessible to the user.

If the part or embedded GUI application is defined inside another visual part, that part is called its *parent part*. The Layout settings describe the position of the part's (or embedded GUI application's) edges relative to the parent part, to other parts, called *targets*, defined within the same parent part, or to another edge of itself. In addition no attachment or proportional grid is definable.

GUI PROGRAMMING EXAMPLE: CONTROLLING VISUAL PART LAYOUT AND SIZING	
	<p>The best way to understand how the Layout settings can be used to control the visual set of parts is to try it:</p> <ol style="list-style-type: none"> From the File pull-down menu select New member.... In the New Member pop-up window the GUI radio button and the Open... push button. Add a push button to the bottom of the window. Add a second push button or, if you created it, add the embedded GUI application to the bottom of the window. To embedded GUIEMB1: <ol style="list-style-type: none"> From the Options pull-down menu select Add GUI application.... Select the GUI member from the drop-down list in the Add GUI application window. Click on the OK push button (your cursor symbol becomes a crosshair and move crosshair to the window part and click mouse button 1. The embedded GUI application defined at the spot of the crosshair. Size and align the push button and the embedded GUI application so that they are the bottom area of the window. Add a list box to the window part above the push buttons. Size the list box so that about as wide as the outside edges of the push buttons (see Figure 10). Save the GUI application as WINTST1 Test the GUI application. Resize the window by grabbing the bottom right-hand corner of the window and dragging the corner to a new point on your display. What happens when the window larger or smaller? End the test session by closing the window Resize the window part directly in the GUI application definition window by grabbing bottom right-hand corner of the window and dragging the corner to a new point on a free-form surface. What happens when you make the window larger or smaller? <p>You should now see that visual parts added to a window, by default, do not resize when the window is resized and that the behavior in test mode is the same as when manipulate the window part in the GUI application definition window.</p> <p>Let's change the Layout settings of the visual parts on the window to modify the behavior:</p> <ol style="list-style-type: none"> Resize the window so that you return to the original view (see Figure 10).
	<p>.</p> <p>.</p> <p>.</p>
	<p>.</p> <p>.</p> <p>.</p>
	<ol style="list-style-type: none"> Double click on the list box to open up the settings window. Using the Layout part Settings notebook, change the Edge selection and Attachment type settings for the to these values:
	<p>Top: Parent top edge</p> <p>Bottom: Parent bottom edge</p>

	Left: Parent left edge	Right: Parent right edge
12.	Resize the window part directly in the GUI application definition window by grabbing the bottom right-hand corner of the window and dragging the corner to a new point on a free-form surface. What happens to the list box and the push buttons, when you make the window larger or smaller?	
	We now have the list box sizing with respect to the window, but it can size right to the top of the push buttons! Let's fix that:	
13.	Double-click on the left of the two push buttons to open up the settings window. If this is the embedded GUI application, you will have to use mouse button 2 to open the settings window for the part and select the Open settings option.)	
14.	Using the Layout page of the Settings notebook, change the Edge selection and Attachment type settings for the left push button to these values:	
	Top: Target bottom edge - Target part: List1 Left: Parent left edge	Bottom: No attachment Right: No attachment
15.	Double-click on the right of the two push buttons to open up the settings window. If this is the embedded GUI application, you will have to use mouse button 2 to open up the settings window for the part and select the Open settings option.)	
16.	Using the Layout page of the Settings notebook, change the Edge selection and Attachment type settings for the right push button to these values in this order:	
	1)Bottom: Parent bottom edge 3)Left: Parent left edge	2)Top: No attachment 4)Right: No attachment
	You may need to reposition and line up the push buttons after these changes so that the view is the same as before (see Figure 10).	
17.	Resize the window part directly in the GUI application definition window by grabbing the bottom right-hand corner of the window and dragging the corner to a new point on a free-form surface. What happens to the list box and the push buttons, when you make the window larger or smaller?	
	We now have the list box sizing with respect to the window and using two different techniques the push buttons staying below the list box. Try and change the push button settings so that they stay inside the window.	

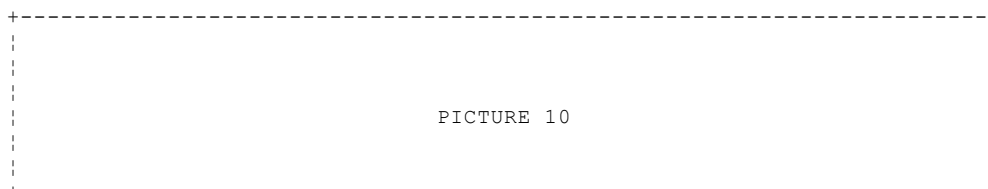


Figure 10. Window with List Box and Push Buttons

Controlling the layout for multiple visual parts in a composite part such as a window or form can be a complex task. Consider the following approaches in your GUI applications:

- ☐ Use embedded GUI applications to build smaller parts.
- ☐ Group visual parts in forms and GUI builders and then focus on managing the layout of these composite parts.
- ☐ Consider using scrolling forms to provide some flexibility on window size.
- ☐ Test the visual layout of your GUI applications on all target display resolutions.

Subtopics

1.4.3.1 Fit a Part to the Parent Part

Some parts, such as a notebook or table, look better if the size exactly fits the parent part. Follow these steps to fit a part to the parent window:

1. Double-click on the part to get the Settings notebook of the part.
2. Select the Layout notebook page.
3. Click on the **Top** radio button.
4. You can select the attachment type from the drop-down list. Set the Attachment type to Parent top edge and the Offset to 0 as shown in Figure 11.

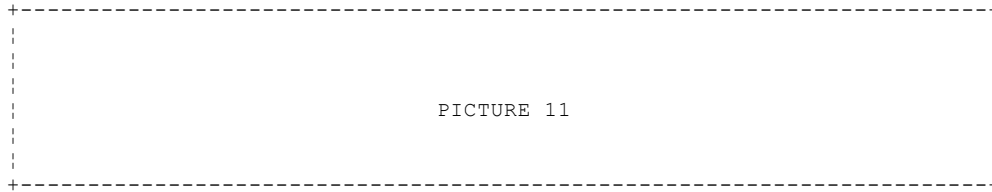


Figure 11. Setting Position of the Top Edge of a Notebook

5. Click mouse button 1 on the **Bottom** radio button and set the Attachment type to Parent bottom edge and the Offset to 0.
6. Set the Attachment type and Offset for the **Left** and **Right** radio buttons to the Parent left and right edge respectively, with an Offset of 0.
7. Click mouse button 1 on the **OK** push button to both apply the setting changes and close the Settings notebook.

You should see the part size and position change such that it is just about the same size as the window.

Try other top/bottom and left/right offset values to see how they affect the appearance of the part in the window.

1.4.3.2 *Distributing Parts Evenly on a Particular Area of a Parent Part*

Suppose you have a window with some text, labels, push buttons, and other parts (see Figure 12 for an example). You want to distribute the text and label parts evenly without disturbing the push buttons and the title label.

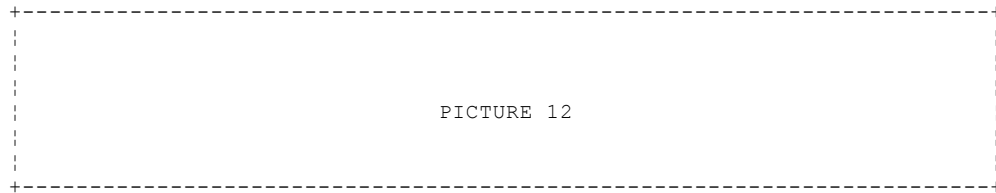


Figure 12. Sample Window with Text and Label Parts to Be Distributed

Here is one way of distributing parts quickly:

1. Drop a form on the text and label parts. Size the form to cover the area that you want to use for the text and label parts (see Figure 13). If you want to visibly group the text and label parts, use a group box instead of a form.

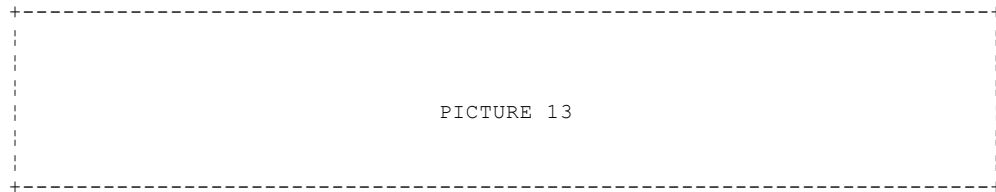


Figure 13. Form Covering the Text and Label Parts

2. Move the form to the free-form surface.
3. Select all text and label parts, drag them, and drop them on the form.
4. Select all text parts and click mouse button 1 on the **distribute vertically** button.
5. Select all label parts and click mouse button 1 on the **distribute vertically** button (see Figure 14). You may need to align the parts as well.

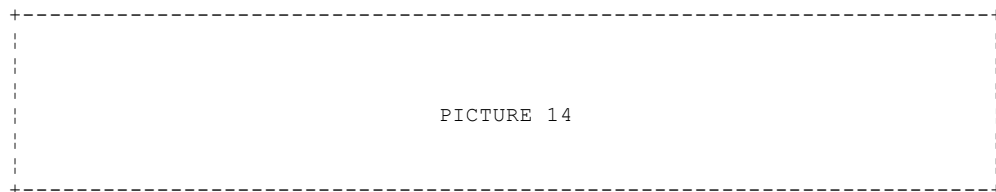


Figure 14. Distributing the Text and Label Parts

6. Select the form, drag it, and drop it back on the window. The text and label parts will come along with the form. We recommend that you move the entire form instead of the texts and labels only. The form is not visible on execution, but it is useful for maintenance.

1.4.4 Summary

There are two fundamental types of windows: the data display window and the data entry window. The function of a window will impact its design. A flexible GUI application system is based on clear, simple, and easy-to-use windows.

The visual view of a GUI applications is based on how you arrange and control the layout of the visual parts.

Windows have options that can control their visual aspects, sizing rules, and where they will be shown on the display. Other composite parts, such as forms and GUI builders, have simple layout options when placed on the free-form surface.

Visual parts placed on other visual parts have powerful layout options that allow you to control the relationships between visual parts on a given composite part during resizing activity.

Layout options allow you to control and manipulate visual parts to fine tune the runtime view of the GUI application.

1.4.5 What You Should Now Be Able to Do

You should now be able to:

- ☐ Understand the differences between data display windows and data entry windows
- ☐ Understand and use the layout control options for visual parts when placed on the free-form surface and on other visual parts
- ☐ Manipulate the visual view of parts, using the layout and distribution options

1.5 Chapter 5. Testing and Debugging GUI Applications

In this chapter we discuss effective ways of testing and debugging your GUI application in both an Interactive Test Facility and runtime mode. We provide an overview of both environments and explain how the execution of the application can be traced and debugged in each.

Subtopics

- 1.5.1 Interactive Test Facility
- 1.5.2 Debugging Hard Errors
- 1.5.3 Runtime Trace
- 1.5.4 Summary
- 1.5.5 What You Should Now Be Able To Do

1.5.1 Interactive Test Facility

ITF includes a number of facilities to analyze your application. You can use testpoints to look at the activity of your application and values for specific variables, and you can use the trace facility to look at both the procedural code that gets executed and the events that get triggered in the GUI application. During execution you can stop an application in a number of ways to look at the results.

Subtopics

- 1.5.1.1 Using Testpoints
- 1.5.1.2 Using Testpoints
- 1.5.1.3 Working with Application Data
- 1.5.1.4 Controlling Test Processing
- 1.5.1.5 Using Trace Log Windows
- 1.5.1.6 Interrupting a Test Run

1.5.1.1 Using Testpoints

ITF provides a number of facilities to analyze the execution of your application. These include:

- ☐ Trace of all actions and events that occur in the system
- ☐ Views of the value of data at a certain point in your applications
- ☐ Breakpoints on statements in processes and statement groups
- ☐ Breakpoints on data items
- ☐ Watchpoints on data items
- ☐ Step-by-step execution of applications, processes, and statement groups
- ☐ Dynamic application partitioning support (3)

The number of facilities you use to analyze your application determines the relative execution speed of an ITF test run.

There are no mechanisms for step-by-step execution of visual connections and for viewing the values that are passed as parameters on a connection.

ITF is started as soon as you test an application. The interface with ITF consists of the Test Monitor window (see Figure 15).

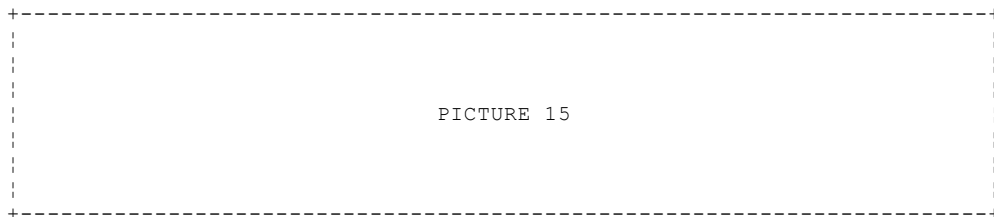


Figure 15. ITF Monitor

The ITF monitor is subdivided into four parts:

Stack Monitor

Each application, process, or statement group that is executed is pushed on the stack. When execution is complete, the entry is popped off the stack. To open the member for editing, double-click on it.

Watchpoint Monitor

The watchpoint monitor shows the current values of variables (data items) that you have chosen to include in the list of watchpoints. You can edit the content of a variable by double-clicking on it.

Statement Monitor

The statement monitor shows the current position within a process or a statement group. You can reposition the statement pointer by clicking on the statement to execute next. You can also set a breakpoint by clicking on a statement with mouse button 2.

Action buttons

Action buttons enable you to control the flow of statement processing. The action button functions, in order from left to right, are:

Step	Runs the highlighted statement in the Statement Monitor window
Leap	Runs the highlighted statement, which will cause the execution of a process, statement group, or called application (or program), without active monitoring of the VisualAge Generator statements involved
Run	Starts a test run at the current relative execution speed setting
Stop	Suspends an active test run
Bypass	Skips over the highlighted statement in the Statement Monitor list
Return	Completes the active process, statement group, or called application, without active monitoring of the VisualAge Generator statements involved. The test run will pause at the statement following the return point of the current application component.

From the View menu you can hide or show the first three parts of

the test monitor.

- (3) We did not study this new feature of VisualAge Generator 2.2 in our project. To learn more, search on the word *partitioning* in the VisualAge Generator Developer help facility.

1.5.1.2 Using Testpoints

Testpoints allow you to control ITF monitoring, tracing, and processing functions. There are several types of testpoints:

Breakpoints	Allow you to suspend a test run
Watchpoints	Provide for direct monitoring of data values
Tracepoints	Control tracing activity

Testpoints can be set by selecting **Options** and then **Set testpoints** from the menu. For records and VisualAge Generator tables you can set breakpoints and watchpoints, for processes, statement groups, and applications you can set breakpoints or tracepoints. Select the member (or EZE word) you want to work with and then the type of testpoint you want to set. Figure 16 shows an example of setting breakpoints and watchpoints on a data item in a record.

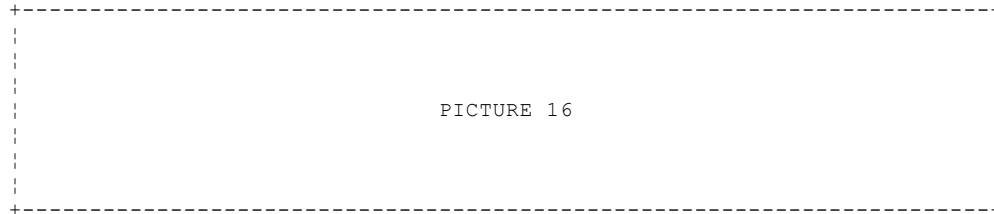


Figure 16. Setting Breakpoints and Watchpoints for a Record

1.5.1.3 Working with Application Data

The ITF also allows you to view and change the data in the application. Choose **Options** and then **Application data...** to get a list of all of the GUI and server applications that are open in the system. Select the application you want to view and select the associated record, VisualAge Generator table, or EZE word for the selected application you want to view. If the current session is read only, you will receive a message indicating that you cannot change the value of the data items.

Depending on the structure of the record or VisualAge Generator table, you will get a view of all of its data items. If a data item has multiple occurrences, you can view them by clicking on the **Occurs...** push button. Choose **Expand...** if the data item is substructured and **Update...** to change the value of the data item. Figure 17 shows an example viewing data items in a working storage record.

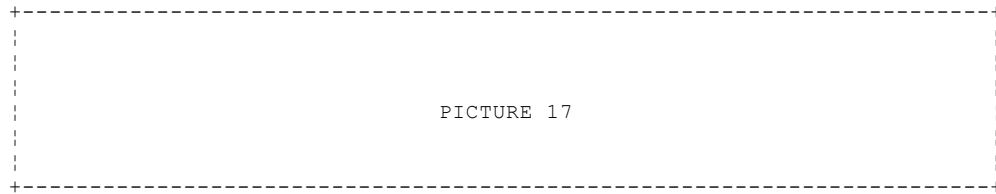


Figure 17. Viewing Data Items in a Working Storage Record

1.5.1.4 Controlling Test Processing

You can change the default characteristics of the ITF by selecting **Profiles** and **Test** and **General preferences...** from any VisualAge Generator Developer menu. These characteristics allow you to control how the ITF session will function. To better understand what the options do, click on the Help push button on the Test Facility - General preferences dialog (see Figure 18).

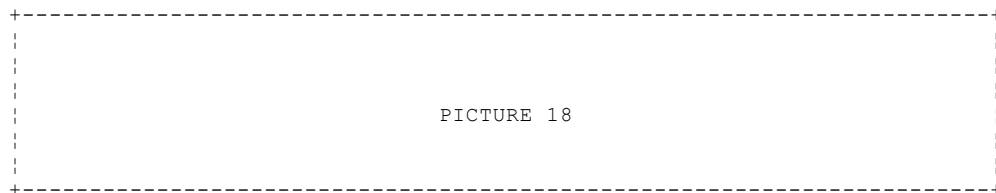


Figure 18. General Preferences for Testing Applications

You can choose to have the ITF implement calls to VisualAge Generator applications as calls to generated server applications through the use of a linkage table. In this case the entire application tree after a call to the first generated application must consist of generated applications. Control returns to the ITF after the completion of the initial call to a generated server application.

If you deselect the **Monitor statements during RUN** toggle button (see Figure 18), you will not see the statements in the statement monitor portion of the Test Monitor window during testing. If you stop testing or reach a breakpoint, the active statements will be visible in the statement monitor area. If you deselect the **Monitor statements during RUN** button, you are reducing the workload on ITF and improving the performance (relative execution speed) of testing.

<u>HINT</u>	
For the fastest possible ITF test run, turn off the Break on event entry option and m	
Test Monitor window. This will allow the ITF to process your logic with minimal over	
approach is excellent for demonstrating code to end users with minimal overhead and w	
having to first generate and prepare the applications for a runtime environment.	

1.5.1.5 Using Trace Log Windows

The trace of the ITF is a powerful mechanism. It enables you to see both the procedural language statements and GUI events that get triggered during execution.

To start a trace, select **Trace** and **Trace all** from the menu of the Test Monitor. All activity will now be written to the trace. If you want to view the trace, choose **Trace** and **View trace...** from the menu. These actions will bring up a window with all of the trace entries. You can open as many of the trace log windows as you want. You can modify the trace options (Trace all, Tracepoints only, Trace nothing) only after you have stopped or paused the testing activity. You can trace the initial activity of a GUI application, the activity that occurs before the first pause, by:

1. Starting test for the GUI application
2. When the test run pauses, choosing **Trace** and **Trace all** from the Test Monitor menu.
3. Choosing **Options** and **Restart test** from the Test Monitor menu.

Now the trace log will contain the initial events that occur in the GUI application.

Generally a trace contains the following type of entries:

GUI event entries

```
00001 GUI Event: sourcePartName (#event) --> targetPartName (#action)
```

These entries contain the events that are signaled throughout the system and cause a certain action to be performed. If the entry concerns an attribute-to-attribute connection, the arrow indicates whether the connection is unidirectional or bidirectional.

GUI event entry for a parameter to a connection

```
00001 GUI Event: parameterPartName (#attribute) -->
                (sourcePartName,event -->
                 targetPartName,action) #attributeOfConnection)
```

These entries indicate the attribute of a part that is being passed as a parameter to a connection. The arrow indicates either the unidirectional or bidirectional character of the connection.

```
+-----+
|  HINT  |
+-----+
```

```
+-----+
|          | A good way of analyzing your application for the inclusion of bidirectional connectio
|          | place a filter on a trace log by using a filtering string of "<-->."
+-----+
```

Application identifier

```
00001 ++++++ MDL00LL 08-22-96 3:19:22 PM ++++++
```

Application identifier entries indicate that a new application has been entered and the rest of the lines correspond to that application until a new entry is found. They indicate the time and date at which the entry was made. These entries are not useful for analyzing the performance of the system because of the overhead involved in using the ITF.

Process identifier

```
00001 ***** MDL00PG-CAL-MDL00U *****
```

Process identifier entries indicate that a new process or statement group has been entered and the rest of the lines correspond to that process or statement group until a new entry is found.

Statement

```
00001 >>>> 027 SET MDL00WL EMPTY;
```

Statement entries indicate the execution of a statement in procedural code. The number represents the line number within the process or statement group.

Application call

```
00001 >>>> 030 CALL MDL00UP MDL00WL, MDL00WS (REPLY;
00002      MDL00WL.ROW(1) = "      "
      MDL00WL.ROW(2) = "      "
      MDL00WL.ROW(3) = "      "
      MDL00WL.MODELID(1) = "    "
      MDL00WL.MODELID(2) = "    "
      MDL00WL.MODELID(3) = "    "
      MDL00WL.MAKE(1) = "      "
      MDL00WL.MAKE(2) = "      "
      MDL00WL.MAKE(3) = "      "
      MDL00WL.MODEL(1) = "      "
      MDL00WL.MODEL(2) = "      "
      MDL00WL.MODEL(3) = "      "
00003      MDL00WS.ROWS-FETCHED = 000
```

Application call entries indicate a call to an application. For each parameter on the call, there is a separate entry indicating the values that were passed.

Switching applications

```
00001      <====> MDL00LL <====> MDL00UP <====>
00002      <==== MDL00LL <==== MDL00UP <====
```

Switching applications entries indicate a switch from one application to another.

Parameter value

```
00001 Info: 'Parameter values:'
00002      MDL00WL.ROW(1) = "      "
      MDL00WL.ROW(2) = "      "
      MDL00WL.ROW(3) = "      "
      MDL00WL.MODELID(1) = "    "
      MDL00WL.MODELID(2) = "    "
      MDL00WL.MODELID(3) = "    "
      MDL00WL.MAKE(1) = "      "
      MDL00WL.MAKE(2) = "      "
      MDL00WL.MAKE(3) = "      "
      MDL00WL.MODEL(1) = "      "
      MDL00WL.MODEL(2) = "      "
      MDL00WL.MODEL(3) = "      "
00003      MDL00WS.ROWS-FETCHED = 000
```

Parameter value entries indicate the parameter values as they are passed to an application.

Switching processes

```
00001      ----> 001 MDL00PU-MAIN ----> 002 MDL00PU-INIT ---->
00002      <---- 001 MDL00PU-MAIN <---- 002 MDL00PU-INIT <----
```

Switching processes entries indicate the control being passed from one process or statement group to another. The direction of the arrows indicates whether the control is passed up or down the application tree.

Assignment

```
00001 >>>> 027 INDEX = 1;
00002      MDL00WA.INDEX = 000
00003      MDL00WA.INDEX = 001
```

If a statement contains an assignment, there are multiple entries that are related to the same line of code. The second line shows the initial value of the data item, and the last line shows the new value of the data item.

Executing a process option


```

00051 >>>> ----- SETINQ MODELS -----;
00052 Before SETINQ...
00053         MODELS.MAKE = "<Not Initialized>"
           MODELS.MODELID = "<Not Initialized>"
           MODELS.PASSENGERS = "<Not Initialized>"
           MODELS.MODEL = "<Not Initialized>"
           MODELS.CLASS = "<Not Initialized>"
00054 CONNECT TO CARRENTL
00055 SELECT
           MODELID, CLASS, MAKE, MODEL, PASSENGERS
        INTO
           :MODELS.MODELID, :MODELS.CLASS,
           :MODELS.MAKE, :MODELS.MODEL,
           :MODELS.PASSENGERS
        FROM
           models T1
        WHERE
           MODELID=> :MODELS.MODELID
        ORDER BY
           1 ASC
00056 After SETINQ...
00057 SQLCODE=0      SQLERRP=SQL02010
           SQLERRD=(0 0 0 0 0 0)      SQLWARN= , , , , , , ,
           SQLEXT= 00000

```

Executing a process option entries indicate the state of the process object before and after the execution of the process option. In some cases an implicit connect to the database is performed.

Testing a condition

```

00001 >>>> 033 WHILE MODELS NOT ERR;
00002         ([MODELS NOT ERR] OK NOT ERR) --> True)

```

Testing a condition entries indicate the test that is performed, the test as it is stated when the data items or EZE words are replaced by their corresponding values, and the result of the test.

A trace log window can have a filter applied to it to indicate that only certain entries should be included. To apply a filter for one trace log window, click on the **Set Filters...** push button on the trace log window. To apply a default trace log filter, select **Profiles** and **Test** and **Default trace entry filters...** from the menu.

Figure 19 shows the Trace Entry Filter window. Table 8 lists the toggle button options in the Trace Entry Filter window.

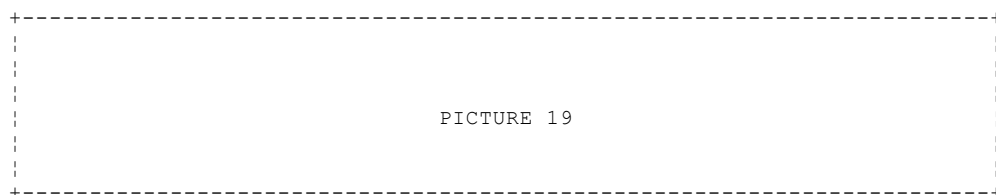


Figure 19. Trace Entry Filter Window

Table 8. Trace Entry Filters	
Filter	Description of Associated Trace Log Window Contents
GUI event	Only lines that contain "GUI event"
DL/1 calls	Only the line containing the actual DL/1 call
SQL statements	Only the lines containing the actual SQL statements
Path labels	Only the lines that contain the indicators of flow between applications, processes, and statement groups
Statements	Only the lines containing the actual procedural statements
File and database I/O	Only the lines that do not contain the actual file or database call but associated with it, such as the movement of data to an SQL row record and the indicators for error information

VisualAge Generator GUI Development Guide

Using Trace Log Windows

Conditional results	Only those lines that contain the results of conditional statements, so
	conditional statements themselves
+-----+-----+	
Parameter data	Only those lines that are related to passing parameter data between app
+-----+-----+	
Results data	Only those lines that are related to the results of assignments or the
	parameters that are passed back to the calling application
+-----+-----+	
Accessed data	Only those lines that contain entries with the values of data accessed
	assignment statements
+-----+-----+	
Error messages	Only lines that contain "Error:"
+-----+-----+	
Warning messages	Only lines that contain "Warning:"
+-----+-----+	
Info messages	Only lines that contain "Info:"
+-----+-----+	
Plus associated statements	Includes all statements that caused other selected trace entry options
	collected
+-----+-----+	
Filter by string	Enables you to include only those entries that contain a certain string
	available in the Default Trace Entry Filters window.
+-----+-----+	

You can have a different set of filter options for each open trace log window. It is quite convenient, for instance, to have a trace log window with just the GUI events and another trace log window with all of the procedural coding.

1.5.1.6 Interrupting a Test Run

During a test cycle sometimes you are required to break up the test run, such as when your application seems to hang or you want to switch to step-by-step monitoring of procedural logic. There are different ways of stopping a test in VisualAge Generator:

Break on event entry

You can select the **Break on event entry** setting through the **Options** menu of the test monitor window. This selection causes the execution to break on every first entry into procedural logic from a GUI application. By default this setting is on when you start ITF the first time.

Breakpoints

You can use breakpoints to stop the execution of a procedural program. With breakpoints you have more control over where the test run is stopped. Breakpoints, once set, can be disabled to allow you to run the application without interruption. The breakpoints can be enabled again when required.

Stop push button

You can click on the stop push button at the bottom of the test monitor. This push button is active only when you are running procedural code. The stop request, once received by the test monitor, will break the test run at the next possible statement.

Sometimes the ITF test run will not hear your stop request. This can occur because the processing is complete before your request was received. You can always reduce the relative execution speed to give your stop request more time to be heard. There are also some rare cases with GUI application where the system input queue is blocked during the processing of the GUI application request. If this is the case, you need to use the Alt+PrintScreen key combination.

Alt+PrintScreen key combination

You can use the Alt+PrintScreen key combination, which causes a user break in the Smalltalk code, to break into GUI applications and other scenarios where the stop request does not produce the desired results. This works most of the time.

The Alt+PrintScreen key combination can also help you get out of situations where your application seems to "hang."

If you use the Alt+PrintScreen key combination the program will be interrupted, and the ITF will try to restore its state and write out a log file according to the settings of your system (see "Debugging Hard Errors" in topic 1.5.2). ITF will use a pop-up window to ask you whether you want to exit. If you click on **Yes**, the VisualAge Generator Smalltalk image will be broughtdown causing all of your open GUI application definitions to close and an error indicating that EZE2IMG ended abnormally. If you click on **No**, you will be able to continue normally. If the reason you had to break the test was because your application was hung up, it may be advisable to restart your test with additional tracing and a reduced relative execution speed.

The system polls for the Alt+PrintScreen key combination. To guarantee that the polling succeeds, hold down both keys until you hear a beep.

The easiest way to ensure control of your testing cycle is to keep the Break on event entry option enabled. If you get tired of telling the ITF to continue, define specific breakpoints.

+-----+
| HINT

	When an error occurs in the ITF, respond to the question of whether you want to exit
	Look at the trace to see where the application stopped. In this way you do not have
	the WALKBACK.LOG file.

1.5.2 Debugging Hard Errors

When an error occurs in your application, VisualAge Generator indicates that it has written out information concerning the error to a file called WALKBACK.LOG. The file is created if you tell your system you want to run your VisualAge Generator applications in debug mode.

In this section we look at how you can set up your system to write out error information to the WALKBACK.LOG file and how to read the information to determine why your GUI application failed.

HINT	
	<p>VisualAge Generator also writes out error information to the TSCRIPT.LOG file when you turn up debug support. Often this is the best place to look first. The TSCRIPT.LOG file contains an entry for the connection that failed. This may be enough to point you to the problem in your GUI application without having to read the WALKBACK.LOG file.</p> <p>You cannot edit the TSCRIPT.LOG file with the E command while VisualAge Generator is running. The file is locked. Use the EPM command instead--but do not save the file.</p>

Subtopics

- 1.5.2.1 Setting Up Debug Mode
- 1.5.2.2 Reading the WALKBACK.LOG

1.5.2.1 Setting Up Debug Mode

To set up debug mode set the following environment variables for the session in which you are running your VisualAge Generator application:

SET EZERDEBUG=anything

This setting causes applications that are tested in the ITF to be in debug mode and write out information to the WALKBACK.LOG file if an error occurs or you perform a user break with the Alt+PrintScreen key combination.

SET EZERRUN_DEBUG=anything

This setting causes applications that are tested in runtime to be in debug mode and write out information to the WALKBACK.LOG file if an error occurs.

The WALKBACK.LOG file is written to the directory set by the EZERTEMP environment variable.

1.5.2.2 Reading the WALKBACK.LOG

To better understand how to read a WALKBACK.LOG file, you first have to understand its basic structure and content. We have created two small GUI applications with errors to use as examples for how to read a WALKBACK.LOG file.

The first GUI application contains a window with a push button. The clicked event of the push button is connected to the atIndex: action of an ordered collection on the free-form surface of the GUI application. A parameter of 0 has been provided as part of the connection between the push button and the ordered collection. When we save and test the GUI application and then click on the push button, an error occurs.

When a problem occurs during GUI application processing in either ITF or runtime, an error window is shown (see Figure 20) and a stack dump is written to the WALKBACK.LOG file.

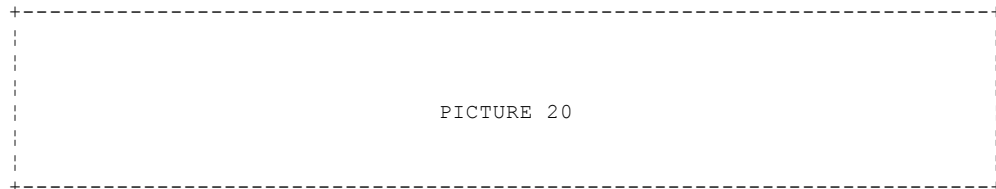


Figure 20. Walkback Error Dialog for Index Out of Range Error

The WALKBACK.LOG file can contain multiple stack dumps, each of which is delimited by blank lines. (4)

To find the last stack dump scan the WALKBACK.LOG file (Figure 21) from bottom up until you find a blank line (1). Then look forward in that particular dump for the word "Error:" or the line that matches the topmost line in the stack dump, which is the error that occurred (2). Everything above this point is related to the processing the error handler does to log the error and put up the message and can be ignored. However, the first line after the word "Error:" is the top of the stack when the problem occurred. Remember that this is a stack dump, and the deeper (further down) you go the earlier in time you are traveling (within the current stack).

All of the entries in the WALKBACK.LOG file are Smalltalk lines of code. They first state the Smalltalk object involved followed by the action that was performed on the object. The lines that follow indicate the values for any parameters, if relevant.

The lines after this first indication of the error explain the message that was sent to a part that caused the error.

As we work down in the stack we see that in Smalltalk you can send any message to any part, but there is the chance that the part you are sending it to does not understand the message. This will result in a *doesNotUnderstand:* situation (3).

The message that is not understood is a connection that was visually built by you or was caused by procedural logic in a process or statement group. The first meaningful indicator of this message will be the line in the WALKBACK.LOG file that includes the message that was sent.

This can be found by first looking further down for an indication that something went wrong. This is identified by *Exception: (ExUserBreak) A break has occurred.* (4).

If we continue down in the stack we can see that the connection involved was clicked of a push button to atIndex of an ordered collection with an index of 0 provided as the parameter. (5).

```
+-----+
|
|      EaRuntimeStartUp class(EsWindowSystemStartUp class)>>#messageLo|p
|      temp1 = 704822
|      temp2 = a CwAppContext
|      [optimized] in UndefinedObject(UIProcess class)>>#forkUserInter|ace
|      UIProcess(Process)>>#newProcessOn:stackSize:
|      arg1 = [] in UIProcess class>>#forkUserInterface
|      arg2 = 1024
|
|      1
|      Walkback at 2:18:53 PM on 11-01-96
|      (ExCLDTIndexOutOfRange) Index out of range.
|      EaRuntimeStartUp class(EsImageStartUp class)>>#outputWalkback:o|:process:
|      arg1 = '(ExCLDTIndexOutOfRange) Index out of range.'
|      arg2 = a CfsWriteFileStream
|
+-----+
```

```

arg3 = UIProcess: (11-01-96 2:14:37 PM) {running, 3}
temp1 = 43
temp2 = EaRuntimeStartUp
temp3 = EsImageStartUp class>>#outputWalkback:on:process:
temp4 = nil
temp5 = 3
temp6 = 10
temp7 = 10
temp8 = false
temp9 = 0
temp10 = 10
[optimized] in EaRuntimeStartUp class (EsWindowSystemStartUp class)>>#outputWalkback:process:
BlockContextTemplate (Block)>>#when:do:exitWith:retryReturn:
arg1 = an ExceptionalEventCollection
arg2 = [] in EsWindowSystemStartUp class>>#outputWalkback:process:
arg3 = [] in Block>>#when:do:
arg4 = an Object
temp1 = [] in Block>>#when:do:exitWith:retryReturn:
temp2 = [] in Block>>#when:do:exitWith:retryReturn:
temp3 = nil
BlockContextTemplate (Block)>>#when:do:
arg1 = an ExceptionalEventCollection
arg2 = [] in EsWindowSystemStartUp class>>#outputWalkback:process:
temp1 = an Object
temp2 = nil
EaRuntimeStartUp class (EsWindowSystemStartUp class)>>#outputWalkback:process:
arg1 = '(ExCLDTIndexOutOfRange) Index out of range.'
arg2 = UIProcess: (11-01-96 2:14:37 PM) {running, 3}
temp1 = 'D:\EZERDEV2\TEMP\walkback.log'
temp2 = -1
temp3 = a CfsWriteFileStream
EaRuntimeStartUp class (EsImageStartUp class)>>#outputWalkback:
arg1 = '(ExCLDTIndexOutOfRange) Index out of range.'
2 EaRuntimeStartUp class>>#reportError:resumable:startBP:
arg1 = '(ExCLDTIndexOutOfRange) Index out of range.'
arg2 = false
arg3 = 320
UIProcess (Process)>>#reportError:resumable:
arg1 = '(ExCLDTIndexOutOfRange) Index out of range.'
arg2 = false
temp1 = 320
[optimized] in UndefinedObject (ExceptionalEvent class)>>#initializeSystemExceptions
blockarg1 = a Signal

Signal>>#evaluate:
arg1 = [] in ExceptionalEvent class>>#initializeSystemExceptions
ExceptionalEvent>>#applyDefaultHandler:
arg1 = a Signal
temp1 = Exception: (ExCLDTIndexOutOfRange) Index out of range
Signal>>#handlesByDefault
[optimized] in BlockContextTemplate (Block)>>#whenOneOf:doMatch:arg:exitWith:retryReturn:
blockarg1 = a Signal
blocktemp1 = nil
[optimized] in BlockContextTemplate (Block)>>#whenExceptions:do:
blockarg1 = a Signal
ExceptionalEvent>>#signalWithArguments:
arg1 = (1)
temp1 = a Signal
temp2 = [] in Block>>#whenExceptions:do:
ExceptionalEvent>>#signalWith:
arg1 = 1
OrderedCollection>>#at:
arg1 = 0
OrderedCollection (SequenceableCollection)>>#atIndex:
arg1 = 0
Message>>#sendTo:
arg1 = OrderedCollection()
3 AbtObservableObjectPartWrapper>>#doesNotUnderstand:
arg1 = a Message
Symbol>>#abrSendTo:withArguments:
arg1 = OrderedCollection()
arg2 = (0)
AbtActionSpec>>#performIn:arguments:
arg1 = OrderedCollection()
arg2 = (0)
AbtEventToActionConnection>>#primSendWithArguments:
arg1 = (0)
AbtEventToActionConnection (AbtEventConnection)>>#send
temp1 = (0)
temp2 = 1
temp3 = nil
temp4 = 1
temp5 = (0)
AbtEventToActionConnection>>#send

```

```

[optimized] in BlockContextTemplate(AbtPart)>>#callHandlers:
BlockContextTemplate(Block)>>#whenExceptions:do:
  arg1 = Exception: (ExError) An error has occurred.
  arg2 = [] in AbtPart>>#callHandlers:
  temp1 = [] in Block>>#whenOneOf:doMatching:exitWith:retryReturn:
  temp2 = [] in Block>>#whenExceptions:do:
  temp3 = nil
BlockContextTemplate(Block)>>#whenErrorExceptionDo:
  arg1 = [] in AbtPart>>#callHandlers:
[optimized] in AbtPushButtonView(AbtPart)>>#callHandlers:

BlockContextTemplate(Block)>>#whenOneOf:doMatching:exitWith:retryReturn:
  arg1 = (Exception: A user error has been signaled. Exception: A user informational
  <...> exception has been signaled. Exception: (ExUserBreak) A break has occurred.)
  arg2 = ([] in Object)>>#exceptionHandlerForUserError [] in Object>>#exceptionHandlerForUserInfo
  <...> in Object>>#exceptionHandlerForUserBreak)
  arg3 = [] in Block>>#whenOneOf:doMatching:
  arg4 = an Object
  temp1 = [] in Block>>#whenOneOf:doMatching:exitWith:retryReturn:
  temp2 = [] in Block>>#whenOneOf:doMatching:exitWith:retryReturn:
  temp3 = nil
BlockContextTemplate(Block)>>#whenOneOf:doMatching:
  arg1 = (Exception: A user error has been signaled. Exception: A user informational
  <...> exception has been signaled. Exception: (ExUserBreak) A break has occurred.)
  arg2 = ([] in Object)>>#exceptionHandlerForUserError [] in Object>>#exceptionHandlerForUserInfo
  <...> in Object>>#exceptionHandlerForUserBreak)
  temp1 = an Object
  temp2 = nil
BlockContextTemplate(Block)>>#when:do:when:do:when:do:
  arg1 = Exception: A user error has been signaled.
  arg2 = [] in Object>>#exceptionHandlerForUserError
  arg3 = Exception: A user informational exception has been signaled.
  arg4 = [] in Object>>#exceptionHandlerForUserInfo
4   arg5 = Exception: (ExUserBreak) A break has occurred.
  arg6 = [] in Object>>#exceptionHandlerForUserBreak
AbtPushButtonView(Object)>>#exceptionHandlersDo:
  arg1 = [] in AbtPart>>#callHandlers:
  [] in AbtPushButtonView(AbtPart)>>#callHandlers:
  arg1 = OrderedCollection(Push Button1 (#clicked) --> Ordered Collection1 (#atIndex:))
  blockarg1 = Push Button1 (#clicked) --> Ordered Collection1 (#atIndex:)
5   BlockContextTemplate>>#apply:from:to:
  arg1 = (Push Button1 (#clicked) --> Ordered Collection1 (#atIndex:))
  arg2 = 1
  arg3 = 1
  temp1 = 1
OrderedCollection>>#do:
  arg1 = [] in AbtPart>>#callHandlers:
AbtPushButtonView(AbtPart)>>#callHandlers:
  arg1 = OrderedCollection(Push Button1 (#clicked) --> Ordered Collection1 (#atIndex:))
[optimized] in AbtPushButtonView>>#userClicked:clientData:callData:
CwAppContext>>#processBackgroundGraphicRequests
  temp1 = [] in AbtPushButtonView>>#userClicked:clientData:callData:
  temp2 = 1
  temp3 = true
CwAppContext>>#readAndDispatch
EaRuntimeStartup class(EsWindowSystemStartup class)>>#messageLoop
  temp1 = 704822
  temp2 = a CwAppContext
[optimized] in UndefinedObject(UIProcess class)>>#forkUserInterface
UIProcess(Process)>>#newProcessOn:stackSize:
  arg1 = [] in UIProcess class>>#forkUserInterface
  arg2 = 1024

```

Figure 21. WALKBACK.LOG File for Index Out of Range Error

The second WALKBACK.LOG example is based on two GUI applications. The first is an embedded GUI application with an entry field on a form. The object attribute of the entry field was promoted to the public interface of the embedded GUI application with the name MyPrivateFeature. The second GUI application has a window where the first GUI application has been embedded. An entry field is added to the window and a connection made between the object attribute of the entry field and the embedded GUI application feature MyPrivateFeature. These applications work fine when tested.

To force a walkback we edited the first GUI application, deleted the promoted feature named MyPrivateFeature, and saved this GUI application. Now when we test the first GUI application, an error occurs (see Figure 22), and a stack dump is written to the WALKBACK.LOG file.

PICTURE 21

Figure 22. Walkback Error Dialog for Missing Public Interface Feature Error

The WALKBACK.LOG file can contain multiple stack dumps, each of which is delimited by blank lines.

To find the last stack dump, scan the WALKBACK.LOG file (Figure 23) from bottom up until you find a blank line (A). This is the start of the stack dump.

Note: We have cut out only the stack dump we want, so our start is at the top. In a busy WALKBACK.LOG, you could have quite a few stack dumps.

After the date and time information for the stack dump the error text is listed (B.) If we look down in the stack dump, we can find this text multiple times. Look for the instance of the text in parenthesis (C). The WALKBACK.LOG file may start to make more sense after this point.

If we scan down from the last point (C), we can see text that defines the error and the members involved (D). This information will help in finding and removing the problem in the GUI application.

A

```

Walkback at 4:58:28 PM on 12-06-96
The feature MyPrivateFeature does not exist in part WALKB2. B
EaRuntimeStartUp class(EsImageStartUp class)>>#outputWalkback:on:process:
  arg1 = 'The feature MyPrivateFeature does not exist in part WALKB2.'
  arg2 = a CfsWriteFileStream
  arg3 = UIProcess:(12-06-96 4:45:21 PM){running,3}
  temp1 = 67
  temp2 = EaRuntimeStartUp
  temp3 = EsImageStartUp class>>#outputWalkback:on:process:
  temp4 = nil
  temp5 = 3
  temp6 = 10
  temp7 = 10
  temp8 = false
  temp9 = 0
  temp10 = 10
[optimized] in EaRuntimeStartUp class(EsWindowSystemStartUp class)>>#outputWalkback:process:
BlockContextTemplate(Block)>>#when:do:exitWith:retryReturn:
  arg1 = an ExceptionalEventCollection
  arg2 = [] in EsWindowSystemStartUp class>>#outputWalkback:process:
  arg3 = [] in Block>>#when:do:
  arg4 = an Object
  temp1 = [] in Block>>#when:do:exitWith:retryReturn:
  temp2 = [] in Block>>#when:do:exitWith:retryReturn:
  temp3 = nil
BlockContextTemplate(Block)>>#when:do:
  arg1 = an ExceptionalEventCollection
  arg2 = [] in EsWindowSystemStartUp class>>#outputWalkback:process:
  temp1 = an Object
  temp2 = nil
EaRuntimeStartUp class(EsWindowSystemStartUp class)>>#outputWalkback:process:
  arg1 = 'The feature MyPrivateFeature does not exist in part WALKB2.'
  arg2 = UIProcess:(12-06-96 4:45:21 PM){running,3}
  temp1 = 'D:\EZERDEV2\TEMP\walkback.log'
  temp2 = -1
  temp3 = a CfsWriteFileStream
EaRuntimeStartUp class(EsImageStartUp class)>>#outputWalkback:
  arg1 = 'The feature MyPrivateFeature does not exist in part WALKB2.'
EaRuntimeStartUp class>>#reportError:resumable:startBP:
  arg1 = 'The feature MyPrivateFeature does not exist in part WALKB2.'
  arg2 = false
  arg3 = 548
UIProcess(Process)>>#reportError:resumable:
  arg1 = 'The feature MyPrivateFeature does not exist in part WALKB2.'
  arg2 = false
  temp1 = 548
[optimized] in UndefinedObject(AbtBaseApp class)>>#setDefaultHandlerForExError
blockarg1 = a Signal
[] in AbtBaseApp class>>#setDefaultHandlerForExError
  temp1 = [] in AbtBaseApp class>>#setDefaultHandlerForExError
  blockarg1 = a Signal
Signal>>#evaluate:
  arg1 = [] in AbtBaseApp class>>#setDefaultHandlerForExError

```

```

ExceptionalEvent>>#applyDefaultHandler:
  arg1 = a Signal
  temp1 = Exception: (ExError) An error has occurred.
Signal>>#handlesByDefault
[optimized] in BlockContextTemplate(Block)>>#whenOneOf:doMatching:exi With:retryReturn:
  blockarg1 = a Signal
  blocktemp1 = nil
[optimized] in BlockContextTemplate(Block)>>#whenExceptions:do:
  blockarg1 = a Signal
ExceptionalEvent>>#signalWithArguments:
  arg1 = ('The feature MyPrivateFeature does not exist in part WALKB2.') C
  temp1 = a Signal
  temp2 = [] in Block>>#whenExceptions:do:
ExceptionalEvent>>#signalWith:
  arg1 = 'The feature MyPrivateFeature does not exist in part WALKB2.
WALKB2 class(Object)>>#error: X1
  arg1 = 'The feature MyPrivateFeature does not exist in part WALKB2. X2
WALKB2 class(Object class)>>#noSuchAttribute:in: D
  arg1 = #MyPrivateFeature
  arg2 = 'WALKB2'
[optimized] in WALKB2(Object)>>#abtWhenChanged:perform:
False>>#ifTrue:ifFalse:
  arg1 = [] in AbtBasicView>>#abtWhenChanged:ifAbsent:perform:
  arg2 = [] in Object>>#abtWhenChanged:perform:
[optimized] in AbtGroupBoxView(AbtBasicView)>>#abtWhenChanged:ifAbsen:perform:
[optimized] in AbtOrderedDictionary(AbtIndexedDictionary)>>#at:ifAbsen:t:
AbtOrderedDictionary(LookupTable)>>#at:ifAbsent:
  arg1 = #MyPrivateFeature
  arg2 = [] in AbtIndexedDictionary>>#at:ifAbsent:
  temp1 = 123
  temp2 = nil
  temp3 = 156
  temp4 = 136
AbtOrderedDictionary(AbtIndexedDictionary)>>#basicAt:ifAbsent:
  arg1 = #MyPrivateFeature
  arg2 = [] in AbtIndexedDictionary>>#at:ifAbsent:
AbtOrderedDictionary(AbtIndexedDictionary)>>#at:ifAbsent:
  arg1 = #MyPrivateFeature
  arg2 = [] in AbtBasicView>>#abtWhenChanged:ifAbsent:perform:
AbtInterfaceSpec>>#interfaceSpecFeatureNamed:ifAbsent:
  arg1 = #MyPrivateFeature
  arg2 = [] in AbtBasicView>>#abtWhenChanged:ifAbsent:perform:
AbtGroupBoxView class(Behavior)>>#abtInstanceFeatureNamed:ifAbsent:
  arg1 = #MyPrivateFeature
  arg2 = [] in AbtBasicView>>#abtWhenChanged:ifAbsent:perform:
AbtGroupBoxView class(Behavior)>>#abtPrimitiveFeatureNamed:ifAbsent:
  arg1 = #MyPrivateFeature
  arg2 = [] in AbtBasicView>>#abtWhenChanged:ifAbsent:perform:
AbtGroupBoxView(Object)>>#abtFeatureNamed:ifAbsent:
  arg1 = #MyPrivateFeature
  arg2 = [] in AbtBasicView>>#abtWhenChanged:ifAbsent:perform:
  temp1 = nil
AbtGroupBoxView(AbtBasicView)>>#abtWhenChanged:ifAbsent:perform:
  arg1 = #MyPrivateFeature
  arg2 = [] in Object>>#abtWhenChanged:perform:
  arg3 = a DirectedMessage
[optimized] in WALKB2(AbtAppBldrPart)>>#abtWhenChanged:ifAbsent:perfo:m:

[optimized] in AbtOrderedDictionary(AbtIndexedDictionary)>>#at:ifAbsen:t:
AbtOrderedDictionary(LookupTable)>>#at:ifAbsent:
  arg1 = #MyPrivateFeature
  arg2 = [] in AbtIndexedDictionary>>#at:ifAbsent:
  temp1 = 21
  temp2 = nil
  temp3 = 22
  temp4 = 22
AbtOrderedDictionary(AbtIndexedDictionary)>>#basicAt:ifAbsent:
  arg1 = #MyPrivateFeature
  arg2 = [] in AbtIndexedDictionary>>#at:ifAbsent:
AbtOrderedDictionary(AbtIndexedDictionary)>>#at:ifAbsent:
  arg1 = #MyPrivateFeature
  arg2 = [] in AbtAppBldrPart>>#abtWhenChanged:ifAbsent:perform:
AbtInterfaceSpec>>#interfaceSpecFeatureNamed:ifAbsent:
  arg1 = #MyPrivateFeature
  arg2 = [] in AbtAppBldrPart>>#abtWhenChanged:ifAbsent:perform:
WALKB2 class(Behavior)>>#abtInstanceFeatureNamed:ifAbsent:
  arg1 = #MyPrivateFeature
  arg2 = [] in AbtAppBldrPart>>#abtWhenChanged:ifAbsent:perform:
WALKB2 class(AbtAppBldrPart class)>>#abtPrimitiveFeatureNamed:ifAbsen:t:
  arg1 = #MyPrivateFeature
  arg2 = [] in AbtAppBldrPart>>#abtWhenChanged:ifAbsent:perform:
WALKB2(Object)>>#abtFeatureNamed:ifAbsent:
  arg1 = #MyPrivateFeature

```

```

    arg2 = [] in AbtAppBldrPart>>#abtWhenChanged:ifAbsent:perform:
    temp1 = nil
WALKB2 (AbtAppBldrPart)>>#abtWhenChanged:ifAbsent:perform:
    arg1 = #MyPrivateFeature
    arg2 = [] in Object>>#abtWhenChanged:perform:
    arg3 = a DirectedMessage
WALKB2 (Object)>>#abtWhenChanged:perform:
    arg1 = #MyPrivateFeature
    arg2 = a DirectedMessage
AbtAttributeToAttributeConnection>>#connect
AbtAttributeToAttributeConnectionBuilder>>#makeConnectionIn:
    arg1 = an AbtBuildPartsObjectSpace
[optimized] in AbtAppBldrPartBuilder>>#hptBuildInternalsOf:objectSpace:
    blockarg1 = an AbtAttributeToAttributeConnectionBuilder
OrderedCollection>>#do:
    arg1 = [] in AbtAppBldrPartBuilder>>#hptBuildInternalsOf:objectSpace:
AbtOrderedDictionary>>#do:
    arg1 = [] in AbtAppBldrPartBuilder>>#hptBuildInternalsOf:objectSpace:
AbtAppBldrPartBuilder>>#hptBuildInternalsOf:objectSpace:
    arg1 = a WALKB1
    arg2 = an AbtBuildPartsObjectSpace
    temp1 = OrderedCollection()
    temp2 = nil
    temp3 = #primaryPart -> an AbtInternalSubpartBuilder(Window1)
AbtAppBldrPartBuilder (AbtTopLevelPartBuilder)>>#hptCreatePart:withInstanceInterfaceSpec:
    arg1 = an AbtBuildPartsObjectSpace
    arg2 = an AbtInterfaceSpec
    temp1 = a WALKB1
HptAppBldrRecord>>#createPart:withInstanceInterfaceSpec:
    arg1 = an AbtBuildPartsObjectSpace
    arg2 = an AbtInterfaceSpec
    temp1 = nil

HptAppBldrRecord (AbtAppBldrRecord)>>#createPartWithInstanceInterfaceSpec:
    arg1 = an AbtInterfaceSpec
CspGUIClientDefinition>>#newPart
    temp1 = nil
    temp2 = an AbtInterfaceSpec
    temp3 = a HptAppBldrRecord
WALKB1 class (CspGuiClientPart class)>>#new
    temp1 = a CspGUIClientDefinition
[optimized] in CspGuiClientPart class>>#testGuiNamed:
BlockContextTemplate (Block)>>#whenExceptions:do:
    arg1 = Exception: (ExError) An error has occurred.
    arg2 = [] in CspGuiClientPart class>>#testGuiNamed:
    temp1 = [] in Block>>#whenOneOf:doMatching:exitWith:retryReturn:
    temp2 = [] in Block>>#whenExceptions:do:
    temp3 = nil
BlockContextTemplate (Block)>>#whenErrorExceptionDo:
    arg1 = [] in CspGuiClientPart class>>#testGuiNamed:
[optimized] in CspGuiClientPart class>>#testGuiNamed:
BlockContextTemplate (Block)>>#whenOneOf:doMatching:exitWith:retryReturn:
    arg1 = (Exception: A user error has been signaled. Exception: A user informational exception has ...
    ... been signaled. Exception: (ExUserBreak) A break has occurred.)
    arg2 = ([[] in Object>>#exceptionHandlerForUserError [] in Object>>#exceptionHandlerForUserInfo ...
    ... [] in Object>>#exceptionHandlerForUserBreak)
    arg3 = [] in Block>>#whenOneOf:doMatching:
    arg4 = an Object
    temp1 = [] in Block>>#whenOneOf:doMatching:exitWith:retryReturn:
    temp2 = [] in Block>>#whenOneOf:doMatching:exitWith:retryReturn:
    temp3 = nil
BlockContextTemplate (Block)>>#whenOneOf:doMatching:
    arg1 = (Exception: A user error has been signaled. Exception: A user informational exception has ...
    ... been signaled. Exception: (ExUserBreak) A break has occurred.)
    arg2 = ([[] in Object>>#exceptionHandlerForUserError [] in Object>>#exceptionHandlerForUserInfo...
    ... [] in Object>>#exceptionHandlerForUserBreak)
    temp1 = an Object
    temp2 = nil
BlockContextTemplate (Block)>>#when:do:when:do:when:do:
    arg1 = Exception: A user error has been signaled.
    arg2 = [] in Object>>#exceptionHandlerForUserError
    arg3 = Exception: A user informational exception has been signaled.
    arg4 = [] in Object>>#exceptionHandlerForUserInfo
    arg5 = Exception: (ExUserBreak) A break has occurred.
    arg6 = [] in Object>>#exceptionHandlerForUserBreak
CspGuiClientPart class (Object)>>#exceptionHandlersDo:
    arg1 = [] in CspGuiClientPart class>>#testGuiNamed:
[optimized] in CspGuiClientPart class>>#testGuiNamed:
CwAppContext>>#processBackgroundGraphicRequests
    temp1 = [] in CspGuiClientPart class>>#testGuiNamed:
    temp2 = 1
    temp3 = true
CwAppContext>>#readAndDispatch
EaRuntimeStartUp class (EsWindowSystemStartUp class)>>#messageLoop

```

```

temp1 = 872138
temp2 = a CwAppContext
[optimized] in UndefinedObject(UIProcess class)>>#forkUserInterface
UIProcess(Process)>>#newProcessOn:stackSize:
  arg1 = [] in UIProcess class>>#forkUserInterface
  arg2 = 1024

```

Figure 23. WALKBACK.LOG File for Missing Public Interface Feature Error

Another way to approach reading the WALKBACK.LOG file (see Figure 23) is to use one of these two approaches:

- ☐ Look for the "error:" string (X1).
- ☐ Using the line that appears at the top of the stack, (B) look for the last occurrence of that line (X2).

From these case studies you should now have an appreciation of the value of the WALKBACK.LOG file. In it you can not only find the connection that was at fault but also some more information about the context within which the error occurred. There is a lot of information in the file that is not relevant to you as a developer, but if you look for the key lines that were mentioned in the description you will be able to better analyze any problems that you encounter in running your applications.

- (4) You may notice that the data in a WALKBACK.LOG file for one error is there twice. When you scan back from the bottom and stop at the first blank line, you have found the second of two stacks written for the error. Only this stack is shown in Figure 21. This stack will help you find the problem in your GUI application.

If you had a complete WALKBACK.LOG file and were to continue to scan back to the next blank line, you will see the first of the two stacks written for the error. The fact that there are two stacks for the same error may actually be a bug, so if you do not see two stacks, it might be that the bug was fixed.

1.5.3 Runtime Trace

Although in runtime you can have any errors dumped to the WALKBACK.LOG file, it is useful to run a trace for situations that do not cause errors, but that do not work as they should and for getting information on the runtime performance of your GUI application.

Subtopics

1.5.3.1 Setting up Profile Mode

1.5.3.2 Reading the TSCRIPT.LOG Profile Data

1.5.3.3 Information about GUI Load Time

1.5.3.1 Setting up Profile Mode

VisualAge Generator writes out an entire trace of the execution of the application to the TSCRIPT.LOG file if you set the profile option to ON with this command:

```
EZE2RUN PROFILE ON
```

After you have run your application, you set the profile option to OFF with this command:

```
EZE2RUN PROFILE OFF
```

The TSCRIPT.LOG file is written to the directory set by the EZERTEMP environment variable. It is not written out until you set the profile option to OFF.

The debug setting must be defined for the profile option to function (see "Setting Up Debug Mode" in topic 1.5.2.1).

1.5.3.2 Reading the TSCRIPT.LOG Profile Data

The TSCRIPT.LOG file can contain multiple entries, with the latest entries at the bottom of the file. The TSCRIPT.LOG file contains three sections when the profile option has been used to trace GUI application runtime processing:

Loading Applications

```
Loading APPNAME from APPNAME.app
XYZZY APPNAME (07-26-96 8:33:44 AM)...
...D:\ezerdev2\EZE2IMG (07-16-96 3:00:36 PM)
```

The loading applications section shows the member that is loaded, the date and time of its creation, and the VisualAge Generator runtime image in which it was loaded.

Event History

```
00001 GUI Event: sourcePartName (#event) --> targetPartName (#action)
00002 GUI Event: parameterPartName (#attribute) --> ...
... (sourcePartName,event --> ...
... targetPartName,action)#attributeOfConnection)
```

The event history section lists all events that have occurred in the system and is identical to a trace in the ITF. (5)

Event Profile

```
ElapsedTime: 54 #Signals: 2 Sequence: (171 230) OnStackCount: 1
(<unnamed>,event --> <unnamed>,action)
```

The event profile section also lists all the events that have occurred in the system, but now in descending order according to the amount of time they took to complete. The elapsed time is the time it took to execute the event (in milliseconds). The elapsed time can include user think time if the event is the first event in a sequence initiated by a user action.

The number of signals is how many times the event was found in the event history listing (which is the first section of the profile output). The sequence collection identifies the position of each event that was signaled in the event history (there is a sequence identifier for each entry in the event history on the left). OnStackCount indicates the depth of connections from where this event was signaled.

The TSCRIPT.LOG file can be useful when you are trying to analyze the performance problems for a GUI application. A full discussion of performance is available in Chapter 13, "Performance" in topic 2.4.

- (5) There seems to be a problem because not all parts listed in the profile data have names. We have reported this as a bug to the lab.

1.5.3.3 Information about GUI Load Time

You can also get runtime information about the load times associated with each GUI application. Use the EZERBENCH environment variable as you used the EZERDEBUG environment variable or set the bench option to ON using this command:

```
EZE2RUN BENCH ON
```

This information is also written to the TSCRIPT.LOG and is part of the first part of the file concerning the loading of applications:

```
loadApplication - APPNAME: 187  
CspGuiClientPart>>#openWidget APPNAME 1176  
CspGuiClientPart class>>#openGuiFromFile: APPNAME.app - 1630
```

The file shows the breakdown of the loading of the application and how much time (in milliseconds) was associated with each load. The debug setting must be defined for the bench option to function (see "Setting Up Debug Mode" in topic 1.5.2.1).

1.5.4 Summary

The VisualAge Generator ITF enables you to run applications without having to generate or prepare them first. Once you are satisfied with your application, you can generate it, producing a .APP file as output. When started with the EZE2RUN command, the .APP file is used with the VisualAge Generator runtime image, to implement the function of the GUI application.

Within the ITF you can analyze the behavior of your application, using testpoints and looking at the contents of data. The ITF also provides a trace of all actions that occurred in the application. You can control your view of this trace information with trace log filters.

When you want to stop the execution of a test run, you can interrupt the procedural logic with the stop push button or press the Alt+PrintScreen key combination.

You can analyze the cause of errors in a GUI application, in both the ITF and runtime, by looking at the WALKBACK.LOG file, which contains information about the conditions in which the error occurred. The WALKBACK.LOG file is written out when you set the EZERDEBUG and EZERRUN_DEBUG environment variables.

In runtime you can also get a trace of all events that occurred in the system. To get this trace you execute the command **EZE2RUN PROFILE ON**. The TSCRIPT.LOG file contains a chronological account of the events that occurred during execution of the application as well as a listing of these events in descending order according to the amount of time they took to complete.

To get information about the load times of GUI applications you either have to set the EZERBENCH environment variable or use the **EZE2RUN BENCH ON** command.

1.5.5 What You Should Now Be Able To Do

You should now be able to:

- ☐ Understand the difference between testing in the ITF and runtime
- ☐ Use the facilities of the Test Monitor to analyze the behavior of your applications
- ☐ Read a trace log
- ☐ Obtain debug information, runtime traces, and information about the load times of GUI applications
- ☐ Read and understand the WALKBACK.LOG file
- ☐ Use the runtime facilities to analyze the behavior of your applications
- ☐ Read and understand the TSCRIPT.LOG file

1.6 Chapter 6. Event-Driven Programming

In this chapter we explain the basic structure of VisualAge Generator GUI applications that enable you to create business and user interface processes by using visual programming techniques. We discuss the implications of an event-driven programming model on the design of your applications.

Subtopics

- 1.6.1 Events As the Basis of Applications
- 1.6.2 Connections
- 1.6.3 Analyzing Events in an Application
- 1.6.4 Designing Event-Driven Applications
- 1.6.5 Summary
- 1.6.6 What You Should Now Be Able to Do

1.6.1 Events As the Basis of Applications

The VisualAge Generator GUI builder is an event-driven programming environment. Event-driven programming languages are built around events that happen or are made to happen throughout the system. The events can be initiated by either the user or the system and can be used as triggers for performing actions. Not all events cause actions to be performed.

The advantage of the event-driven programming approach is that programming is based on the user's interaction with the system. You program behavior for each action that a user can take within the application.

An event can be defined as a *change in the state of the system*. For example, when a user clicks on a push button, the state of the push button changes from being up to being pressed down (clicked). This change in state (event) can be used to trigger an action, such as closing a window.

Events are inherently asynchronous, although the actions that are triggered by one event may be sequenced. This is contrary to procedural programming languages that require you to perform actions in a certain predetermined order. A procedural program has a flow and requires you to ask "What should happen next?" When building an event-driven system, you ask "What should happen when the user clicks on the push button?" This type of question is characteristic of event-driven application environments such as those provided by OS/2 and Windows. Building applications for such environments by providing answers to questions about events is easier than when you have to describe all questions (as you would for a procedural programming language).

1.6.2 Connections

In this section we focus on defining business and user interface processes by using the basic concepts of events and actions of the VisualAge Generator GUI builder.

Subtopics

- 1.6.2.1 Creating Connections
- 1.6.2.2 Types of Connections
- 1.6.2.3 Changing Connections
- 1.6.2.4 Changing the Order of Connections

1.6.2.1 Creating Connections

In the VisualAge Generator GUI builder, the relationship between a certain event and a corresponding action is called a *connection*. Any such connection is shown visually and is represented by a line connecting the parts that are involved (source of the event and target of the action).

GUI PROGRAMMING EXAMPLE: A SIMPLE CONNECTION	
	This example shows how a connection is made between an event and an action.
	1. Create a new GUI application and place a push button on the window with the label
	2. Select the Object menu of the push button by clicking on the push button with mouse button 2.
	3. Select Connect .
	4. Choose <u>clicked</u> from the cascaded menu.
	5. Move the spider to the window and click with mouse button 1.
	6. Choose <u>closeWidget</u> from the menu that appears.
	7. Test the application by clicking on the test button in the tool bar (save the GUI application as CONTST1).
	8. Check whether clicking the push button closes the window.
	The final application is shown in Figure 24.

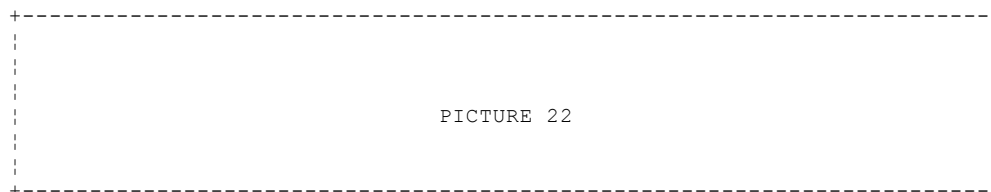


Figure 24. A Simple Event-Driven Application (Stage 1)

1.6.2.2 Types of Connections

Connections are made between the interfaces (also called *features*) of the parts used in the VisualAge Generator GUI builder. Each interface consists of:

- ☐ Actions, which make the part do something
- ☐ Attributes, which are the characteristics of the part
- ☐ Events, which are caused by changes in the state (characteristics) of a part and can trigger actions in other parts or in the part itself

You can see these three elements of the interface of a part when you select the Connect menu of a part.

+-----+ <u>HINT</u> +-----+	
+-----+ +-----+ You can bring up the Connect menu directly by holding down the Alt key and clicking o with mouse button 2. If the part does not have a Connect menu, the Connect window wi the actions, attributes, and events of the part is shown instead. +-----+ +-----+	

To get the Connect window with all features, choose **All features...** from the Connect menu. Figure 25 shows an example of a Connect window with actions, attributes, and events.

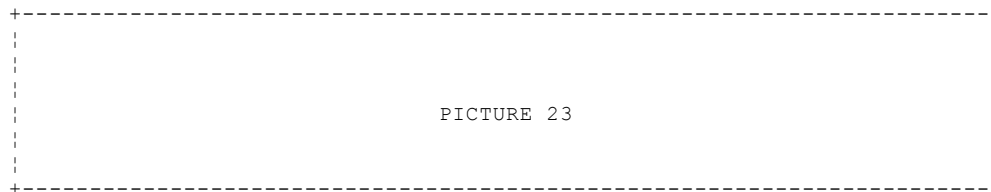


Figure 25. Connect Window of the Window Part

Parts can be connected to each other in a number of ways, each resulting in a different type of connection with different characteristics, as described below. Although we distinguish different types of connections, all connections in VisualAge Generator are essentially event-to-action connections. A change in an attribute is a type of event, and changing an attribute is a type of action.

Subtopics

1.6.2.2.1 Event-to-Action

By choosing an event from the source part and an action from the target part, you create an event-to-action connection. The connection is shown as a green arrow, with the arrowhead pointing to the target part. The connection we created in our previous example between the clicked event of the push button and the closeWidget action of the window is an event-to-action connection.

Some actions expect parameters to be passed to them (see "Providing Parameters" in topic 1.6.2.2.5). These actions can be recognized by the fact that they contain one or more colons in the name. Connections that expect parameters initially show up as dashed lines. Only when the connection has been provided with its parameters does the line become a full line.

1.6.2.2.2 Attribute-to-Attribute

An attribute-to-attribute connection ensures that data for both the source and the target part are kept synchronized. Any changes in the source are propagated to the target and vice versa. This type of connection causes a blue line to be created with pinheads at both ends.

By default, attribute-to-attribute connections are bidirectional, causing changes in either attribute to be propagated to the other. When the GUI application is created, all attribute-to-attribute connections in the GUI application are initialized. The initialization algorithm works as follows:

```
If source value is not nil
  Align target with source      (target takes the value of source)
Else
  If target value is not nil
    Align source with target    (source takes the value of target)
  End
End
```

The source attribute can be made read only by opening the settings of the connection (by either double-clicking on the connection or selecting the Object menu of the connection and choosing **Settings**) and checking the **Read-only source** toggle button. If you want the target end to be read only, you can reverse the connection by clicking on the **Reverse** push button. This action makes the source the target and the target the source. The pinhead at the read-only side of the connection becomes hollow, so you can see the unidirectional character of the connection.

1.6.2.2.3 *Event-to-Attribute*

An event can also be connected to an attribute because changing the value of an attribute is just another type of action. Creating an event-to-attribute connection also causes an arrow with a dashed line to appear. The connection expects the value to be given to the target attribute to be passed as a parameter.

Changing the value of an attribute of a part always causes the state of the part to change, and thus an event is signaled. This change only has consequences if the related event is used to perform some kind of action.

1.6.2.2.4 Attribute-to-Action

An attribute-to-action connection is identical to an event-to-action connection. The change in the value of the attribute represents a change in the state of the part. The change causes an event to be signalled.

Sometimes it helps to think of an attribute as simply a combination of an event and an action. The event is the fact that the attribute changed, and the action is the act of changing the value of the attribute.

HINT
When using a change in an attribute as an event, select it from the list box containing events in the Connect window. This selection ensures that the connection created is attribute-to-attribute connection and will not become one when you change the source of the connection.

1.6.2.2.5 Providing Parameters

Parameters can be provided to a connection either hard coded or by connecting an attribute of a part to the connection.

To provide a hard coded parameter, open the settings of the connection. Click on the **Set parameters...** push button from the Settings window of the connection and define the parameters. If the parameter is incorrect, the entry field for the parameter displays ****error****. Change the value to a valid value to prevent runtime errors. VisualAge Generator does not force you to enter data of a correct type.

HINT
Any parameters that are provided during runtime by connecting an attribute to the parameter will override the initial settings.

Providing a parameter to a connection by using an attribute value of a part is much like creating an attribute-to-attribute connection. Instead of another part being the target of the connection, the connection requiring the parameter is the target. From the Connect window of the part, select the attribute that contains the value for the parameter. If you move the spider over the connection, a square appears on the connection. This is the target for the parameter connection. If you now click mouse button 1, the Connect menu of the connection is displayed. Select the feature to which you want to make the connection. For example, if you are providing the first and last indexes of a range, one of the attributes of the connection would be firstIndex, indicating the starting index of the range.

The Connect menu of a connection always has a result attribute, which contains the value of the result of the connection. For example, if the connection determines the index of a certain item in an array, result would contain the index. Result values of connections are discussed in more detail in "Triggering Events from Other Connections" in topic 1.9.1.2.5.

The data type of the parameter should conform to the expected format. If it does not, a warning is displayed. Sometimes the GUI builder guesses at the data type compatibility because it has insufficient information, so you can sometimes ignore the warnings.

```

GUI PROGRAMMING EXAMPLE:  CONNECTIONS
+-----+
+-----+
+-----+
+-----+

To move the items from one list box to another list box and view the item that is selected in
both windows, do the following:

1. Add the following objects to the window:

    □ Two List Boxes, one of which has a number of default entries (these can be found on the
      first page of the Settings notebook).

    □ A label above the left List Box

    □ An entry field above the right List Box

    □ A push button with ">>" as its label between both List Boxes

    See Figure 26 for the part layout.

2. To see the entry that was selected from the left List Box, connect the selectedItem
   attribute of the leftmost List Box to the object attribute of the label. This
   attribute-to-attribute connection ensures that the selected entry from the left List Box is
   shown in the label.

3. Make the same connection between the right List Box and the entry field above it.

4. Connect the clicked event of the ">>" push button to the items attribute of the right
   List Box. The items attribute contains all of the elements of the List Box. Notice that the
   line that is created is dashed and requires a parameter. The parameter that is passed should
   be of the same type as the target of the connection, in other words, item.

```

5. Connect the items attribute of the left List Box to the value attribute of the co
This will move the data from the left List Box to the right List Box when the pus
clicked.
6. Save the GUI application as **CONTST2**.
7. Test the application. Select an entry from the left List Box. Notice that the te
label changes. Click on the ">>" push button. The entries from the left List Bo
to the right List Box. Select one of these entries. The result shows up in the
field. Now type the text of one of the other items in the List Box in the entry
This item is selected in the list.
8. Use the connection settings window to change the attribute-to-attribute connectio
the right List Box and the entry field to unidirectional (with the List Box as th
and the entry field as the target).
9. Retest the application.

Changing the text in the entry field should no longer influence the selected item
List Box

The final application is shown in Figure 26.

PICTURE 24

Figure 26. A simple event-driven application (stage 2)

1.6.2.3 *Changing Connections*

In this section we look at how you can change connections in your application. We cover viewing the connection, changing the features involved in the connection, changing the source or target of a connection, moving connection lines, and deleting a connection.

Subtopics

- 1.6.2.3.1 Viewing the Connection
- 1.6.2.3.2 Changing the Features
- 1.6.2.3.3 Changing the Source or Target
- 1.6.2.3.4 Moving Connection Lines
- 1.6.2.3.5 Deleting a Connection

1.6.2.3.1 Viewing the Connection

You can view a connection by selecting it. The connection is shown at the bottom of the informational area of the window in the following format:

```
('sourcePartName',#event --> 'targetPartName', #action)
```

When multiple connections are involved (for example, the connection contains a parameter that is set by using an attribute-to-attribute connection), the connections are shown in a nested format:

```
('parameterPartName',#attribute <--> '('sourcePartName',#event -->  
'targetPartName', #action)',#value)
```

For attribute-to-attribute connections, the arrow indicates whether the source is read only or not. A bidirectional arrow indicates a bidirectional connection.

1.6.2.3.2 Changing the Features

You can change the features involved in the connection by opening the settings of the connection (see Figure 27). The window indicates the current connection in the status area. The events and actions involved are initially selected in the left and right list of events and actions. To change the connection, select a new entry from either of the two lists. You can revert to the current setting by clicking on **Show current**.



Figure 27. Settings Window of a Connection

+-----+ <u>HINT</u> +-----+	
+-----+ +-----+ If you are changing the connection from, for example, an event-to-attribute connection event-to-action connection, the list with the target features does not contain the ac the target part. You can type in the action name in the entry field for the target p sure that you type in the name correctly. The VisualAge Generator GUI builder is cas sensitive, and typing mistakes cause runtime errors. +-----+ +-----+	

1.6.2.3.3 Changing the Source or Target

You can change the source or target of a connection by selecting the connection and dragging the end of the connection that has to be changed to a different part. If the new part does not contain the feature to which the connection was originally made, the connect menu of the new source or target part is displayed when you drop the connection. Thus you can change the target action, attribute, or event.

1.6.2.3.4 Moving Connection Lines

When a connection is selected, it shows three black dots within the line: one in the middle and one at each end. You can drag the middle dot to a different location to change the path of the line. The path of the line is adjusted automatically.

Each segment of this line now in turn has three dots, one of which can again be changed to change the path of the segment. This breakdown of lines into segments continues for as long as you need to change the path of the line in more detail, thus enabling you to make a clear visual layout of the connections.

1.6.2.3.5 Deleting a Connection

To delete a connection, select it and either:

Press the Delete key

or

Choose **Delete** from the Object menu.

You can select multiple connections just as you can select multiple parts in VisualAge Generator. Drag over them using mouse button 1 or add individual connections to the selection by holding down the Control key and clicking mouse button 1.

If you delete a part that still has connections attached to it, you will be asked whether you are sure that you want to delete the part. This check is especially useful if you have hidden connections by using the **Hide connections** option from the Tools menu or from the tool bar.

1.6.2.4 Changing the Order of Connections

In general, events are asynchronous. However, when one event starts multiple actions, the actions are executed in a certain sequence. To view the sequence, choose **Reorder connections from...** from a part's Object menu. All connections that lead from the part to other parts and the order in which they are executed are shown. You can change the order by dragging a connection, using mouse button 2, either up or down the list. Figure 28 shows an example of reordering connections.

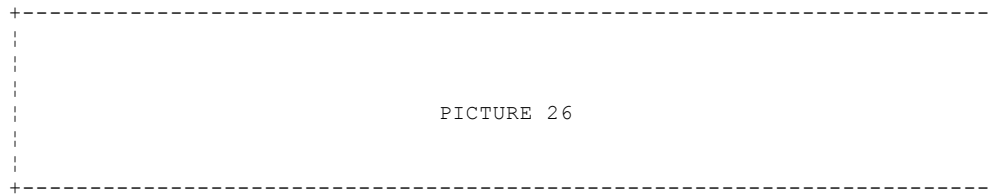


Figure 28. Changing the Order of Connections

It is important to be aware of the order of the actions triggered by the connections. Imagine trying to save data from a window after it is closed instead of before it is closed.

1.6.3 Analyzing Events in an Application

The events that are triggered by the different parts and the sequence in which they occur can tell you a lot about the execution of your application. The ITF can show a trace of all connections that executed within the application. To start the trace, run the application and select **Trace** and then **Trace all** from the menu of the Test Monitor window. You might have to restart the test to get all of the events from the start. To restart the test, select **Options** and then **Restart test** from the menu of the Test Monitor window.

Setting the trace indicator writes out a trace that can be viewed from the Test Monitor window by choosing **Trace** and the **View trace....** The trace contains all of the events that were triggered, in chronological order.

GUI PROGRAMMING EXAMPLE:	EVENT TRACE
	<p>The following event trace (as available in the ITF Trace log window) shows the event-combinations in chronological order that occurred in our application when executed at the script in the previous GUI programming example (see Connections in topic 1.6.2.2).</p> <pre> 00001 GUI Event: List1 (#selectedItem) <--> Label4 (#object) 00002 GUI Event: List1 (#items) <--> (Push Button2,clicked --> List2,items) (#value) 00003 GUI Event: List1 (#selectedItem) <--> Label4 (#object) 00004 GUI Event: Push Button2 (#clicked) --> List2 (#items) 00005 GUI Event: List2 (#selectedItem) <--> Text1 (#object) 00006 GUI Event: List2 (#selectedItem) <--> Text1 (#object) 00007 GUI Event: List2 (#selectedItem) <--> Text1 (#object) 00008 GUI Event: List2 (#selectedItem) <--> Text1 (#object) 00009 GUI Event: List2 (#selectedItem) <--> Text1 (#object) 00010 GUI Event: List2 (#selectedItem) <--> Text1 (#object) 00011 GUI Event: List2 (#selectedItem) <--> Text1 (#object) 00012 GUI Event: Push Button1 (#clicked) --> Window (#closeWidget) </pre> <p>The trace shows the following events:</p> <ol style="list-style-type: none"> 1. Initialization of the attribute-to-attribute connections to parts that have a value entry 1 and 2). To capture these events in the ITF trace log window we had to turn on support and then restart the test run. 2. Selecting the item from the List Box causes the <u>object</u> attribute of the label to be changed to the <u>selectedItem</u> attribute of the List Box (trace entry 3). 3. Clicking the push button causes the items of the first List Box to be copied to the second List Box (trace entry 4). 4. Selecting the item from the right List Box causes the value of the <u>object</u> attribute of the entry field to be changed to the value of the <u>selectedItem</u> attribute of the List Box (trace entry 5). 5. Changing the entry field (trace entries 6-11). Because the Notify change on each keystroke is set for the entry field in its settings, each keystroke causes the attribute-to-attribute connection to be executed. When this attribute is not set, the connection executes only when the entry field loses focus. 6. Clicking the Close push button causes the window to be closed (trace entry 12).

To use the trace effectively it is important that the entries are readable. Having entries with "List1" and "List2" is not very meaningful. It would be better to have "liLeftList" and "liRightList" (see Appendix B, "VisualAge Generator Naming Convention" in topic B.0). The names shown in the trace are equal to the names of the parts. You can change the name of a part through its settings or by using the parts list. The trace from the previous example, now without **Notify change on each keystroke** set, would look like this:

```

00001 GUI Event: liLeftList (#selectedItem) <--> lbLabel (#object)
00002 GUI Event: liLeftList (#items) <--> (pbMove,clicked --> liRightList,items) (#value)
00003 GUI Event: liLeftList (#selectedItem) <--> lbLabel (#object)
00004 GUI Event: pbMove (#clicked) --> liRightList (#items)
00005 GUI Event: liRightList (#selectedItem) --> efLabel (#object)

```

A filter can be set on the trace either permanently or just for the session you are running. The trace file can be seen as a database, and the filter as a way of looking at the data that is there. See "Using Trace Log Windows" in topic 1.5.1.5 for more information about using

filters and the trace log.

HINT
You can open multiple trace windows for a single tested application, each with its own filter applied to it.

1.6.4 Designing Event-Driven Applications

Creating a technical design for an event-driven application system requires a different approach from designing procedural programs and screen-driven 3270 applications. With a procedural program, the programmer is in control and determines what the user can do. With an event-driven program, the user is in control. Instead of determining what the user is allowed to do, the focus shifts to what the user might do and what the program should do in response. Thus your design should concentrate more on user tasks than on functions. The functions are a result of user tasks and should be described as such. The design should also indicate when things should happen.

One of the most important aspects in designing a robust event-driven system is ensuring that it is, at any time, in a stable state. Any action a user takes should bring the system back in a state from which it is possible for him or her to undertake any other action.

In general achieving a stable state requires the development of components that are as independent of each other as possible. Actions should be atomic. They should not allow other actions to interfere with their operation and, in theory, should take no time to execute.

1.6.5 Summary

VisualAge Generator GUI applications are event-driven. Events are caused by changes in the state of a part. They are nonprocedural in nature. It may help to consider them as asynchronous, because the parts coexist simultaneously, even though there is a subtle sequence to a series of events triggered from the same action. This is discussed further in Chapter 9, "Visually Building Logic" in topic 1.9. Both the system and the user can cause events to be triggered.

Events that cause actions are represented as lines between parts in the VisualAge Generator GUI builder. These lines connect an event to an action, an event to an attribute, or an attribute to another attribute. A connection can require a parameter to be passed to it. The actions that occur based on a certain event can be reordered.

The events that are executed by the system can be viewed by looking at the trace of an application. The trace contains a chronological log of all event-to-action and attribute-to-attribute connections.

Designing an event-driven application requires focusing on the things that the user wants to do instead of the things that the application wants the user to do. The application should therefore always be in a reenterable state.

1.6.6 What You Should Now Be Able to Do

This chapter should have given you a basic understanding of event-driven programming. By now you should be able to:

- ☐ Understand event-driven applications
- ☐ Describe the implications of event-driven programming for your design
- ☐ Build a basic event-driven application, using VisualAge Generator
- ☐ See and analyze the events that occur in a system

Using connections to build applications is covered in more detail in Chapter 9, "Visually Building Logic" in topic 1.9.

1.7 Chapter 7. Working with Data in VisualAge Generator

In this chapter we first define the use of data in VisualAge Generator. We look at the basic components that make up structures of data, the different structures of data, and the way these can be used in GUI applications.

The chapter focuses on the use of data in a GUI application; it does not cover all aspects of data related to VisualAge Generator. For more information about data and platform considerations, see the VisualAge Generator manuals.

Subtopics

- 1.7.1 Data Elements
- 1.7.2 Working Storage Records
- 1.7.3 Data Structures
- 1.7.4 Occurs Items
- 1.7.5 VisualAge Generator Tables
- 1.7.6 Ordered Collections
- 1.7.7 Sharing Data
- 1.7.8 Data Format in a GUI Application
- 1.7.9 Summary
- 1.7.10 What You Should Now Be Able to Do

1.7.1 Data Elements

An unstructured, single data element in VisualAge Generator is called a *data item*. The kind of data that is described by a data item is called the *data type* of the data item.

Subtopics

1.7.1.1 Data Items

1.7.1.2 Data Types

1.7.1.3 Defining Data Items

1.7.1.4 Special Considerations for Defining Data Items

1.7.1.1 Data Items

A data item itself is not a part of a GUI application. It can only become accessible within a GUI application if it is embedded in a *record member part* or *table member part*. Data items have a scope of definition. The definition of the data item is either globally shared or local to the structure in which it is used.

A global data item is stored as a VisualAge Generator member. Its definition and characteristics are reusable in multiple records or tables by using its member name in the record or table. Therefore its name must be unique across all member types within the active MSL concatenation and should strictly follow your naming conventions. Also any change to the characteristics of a global data item is reflected in all of the records or tables in which it is used.

A data item that is not global is called a *local data item*. A local data item can be defined only inside a record or table. Local data items are not saved in an MSL but are stored with the record or table in which they were defined. Therefore a data item name need only be unique within a record or table, but for clarity reasons we recommend applying naming conventions to local data items too. Changes to the characteristics of a local data item have no effect on definitions of data items with the same name in other records or tables.

As a general rule, define data items as global only if they are part of your organization's data model. All other data items should be local. This approach will keep the number of VisualAge Generator members defined in your MSLs small, which will enhance the usability and maintainability of your application system. It is also in line with the use of VisualAge Generator in a Computer Aided Software Engineering (CASE) environment where data definitions are driven by a data model. (6)

To avoid application errors caused by defining multiple global data items with the same name and different characteristics and to allow for better control of the definition of global data items, define all global data items in one MSL used in read-only mode by all developers.

- (6) For instance, using DataAtlas and TeamConnection, you can use the definition of a shareable data element on which you can base the definition of elements in your data model and application (global data items). You could define a telephone number to be a shareable data element, with both a home telephone number and work telephone number deriving their definition from the shareable data element. If you create a global data item, it always has to be linked to a shareable data element. If you do not explicitly define one, it is created for you.

1.7.1.2 Data Types

There are two kinds of data types:

Numeric Used to define integer and decimal data

Nonnumeric Used to define data containing any kind of characters

Thus the data type specifies the internal format of the data item and determines how the item is processed when referenced.

Subtopics

1.7.1.2.1 Numeric Data Types

1.7.1.2.2 NonNumeric Data Types

1.7.1.2.1 Numeric Data Types

You can define the following numeric data types:

Bin	The Bin data types support numeric data and store it in binary format. The Bin data types can store large numbers in a smaller number of bytes than other numeric data types.
Num	Num data types support numeric characters that have both <i>F</i> and <i>C</i> as the byte value of the positive sign in EBCDIC. Thus num data types require conversion and are slower.
Numc	Numc data types are identical to Num but only support <i>C</i> as the byte value of the positive sign in EBCDIC. Therefore no conversion occurs.
Pack	Pack data types support packed numeric data, which has two digits in every byte. Pack only supports <i>C</i> as the byte value of the positive sign in EBCDIC and therefore does no conversion.
Pacf	Pacf data types support packed numeric data. Pacf supports both <i>F</i> and <i>C</i> as the byte value of the positive sign in EBCDIC. Therefore it requires conversion and is slower.

That said, generation options (**/POSSIGN**) can be used to control the techniques (*F* or *C*) for identifying the sign in numeric data types. Review *Generating VisualAge Generator Applications* for more information.

1.7.1.2.2 NonNumeric Data Types

You can define the following nonnumeric data types:

Char	Char data types support alphabetic, numeric, or national characters.
DBCS	DBCS data types support double-byte characters. Double-byte character set (DBCS) data is ideographic character data that requires two positions for each character. Double-byte characters are required for the Japanese, Korean, and Chinese languages.
Hex	Hex data types support hexadecimal (base 16) values. Hexadecimal data types provide basic processing functions (moves, comparisons, and parameter passing) for data whose data type VisualAge Generator does not directly support VisualAge Generator.
Mixed	Mixed data types support data items that can contain both single-byte character set (SBCS) and DBCS data. The length specified for a mixed data type is the number of single-byte characters that the field can contain.

Which data type you use depends on the platforms on which the data is used, the database system you use, and performance implications. These considerations are documented in the VisualAge Generator manuals.

1.7.1.3 Defining Data Items

To define a global data item, you create a new member with type **ITEM** in the MSL. Select **File** and **New member...** from the menu and select **Item** as the type of member to be created. Figure 29 shows the Data Item Definition window for defining a global data item.

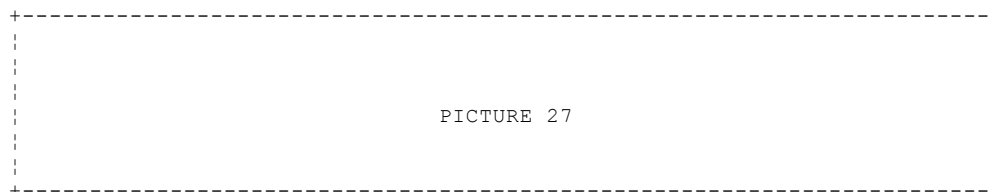


Figure 29. The Data Item Definition Window

On the Data Item Definition window, you define the following characteristics of a data item:

Description

To document the data item, enter a 1- to 30-character case-sensitive description in the **Description** entry field. The description of a data item, if entered, becomes the label of data entry fields, lists, or container columns that are created by using *Quick Form*, a VisualAge Generator function to create default visual parts and labels from other parts. If the data item does not have a description the name of the data item is used instead. See "Quick Form" in topic 1.7.2.2 for more information about using Quick Form to create default entry fields and labels from data items.

Data type

Choose one of the numeric or nonnumeric data types.

How you want to specify the length

Select the **Length** radio button if you want to specify the length in digits. Select the **Bytes** radio button if you want to specify the length in characters required to store the data item internally. Whatever selection you make, VisualAge Generator calculates the other value automatically.

Length of the data item in digits or bytes

Use the **Length or Bytes spin button** to specify the number of digits or characters of the data item. The length of a numeric data item must include the decimal places.

Number of decimal digits

For a numeric data item, use the **Decimal places** spin button to specify the number of digits reserved to the right of an implied decimal point. The default is 0. The number of decimal digits must be less than or equal to the number of digits of the data item.

The allowable size of a data item depends on its use. A data item can be used in either a working storage record or a VisualAge Generator table. Table 9 shows the possible specifications of length, bytes, and decimal places of data items according to the type of structure in which the data item is used.

Table 9. Data Item Specifications						
Data Item	Used in Working Storage Record			Used in VisualAge Generator Table		
	Digits	Bytes	Decimal Digits	Digits	Bytes	
Char	1 to 32767	1 to 32767	N. A.	1 to 254	1 to 254	
DBCS	1 to 16383	2, 4, ..., 32766	N. A.	1 to 127	2, 4, ..., 254	
Hex	2, 4, ..., 65534	1 to 32767	N. A.	2, 4, ..., 254	1 to 127	
Mixed	1 to 32767	1 to 32767	N. A.	1 to 254	1 to 254	
Num/Numc	1 to 18	1 to 18	0 to 18	1 to 18	1 to 18	
Pack/Pacf	1, 3, ..., 17, 18	1 to 10	0 to 18	1, 3, ..., 17, 18	1 to 10	
Bin	4	2	0 to 4	4	2	
	9	4	0 to 9	9	4	
	18	8	0 to 18	18	8	

GUI PROGRAMMING EXAMPLE: DEFINING A GLOBAL DATA ITEM

To create a global data item named ITMTST1 representing numeric data in packed format, 9 digits before and two after the decimal point, do the following:

1. From the **File** pull-down menu, select **New member**.
2. On the New Member pop-up window click on the **Item** radio button and the **Open...** button.
3. Click on the **Pack** radio button.
Because **Pack** is a numeric data type, the **Decimal places** spin button becomes accessible.
4. The **Length** radio button is preselected. Keep it.
5. Enter 11 into the **Length** spin button because the total length of the data item is 11 (9 plus 2).
6. Enter 2 into the **Decimal places** spin button.
7. Toggle between the **Length** and **Bytes** radio buttons. Notice how the values are related.
8. Save the data item as **ITMTST1**.

If you save a global data item with the name of a global data item that already exists in one of the MSLs of the active MSL concatenation, VisualAge Generator displays a warning message. However, you can replace the old global data item if it was defined in your read/write MSL, or you can create a new global data item with the same name if the old global data item was defined in a read-only MSL of the active MSL concatenation.

1.7.1.4 Special Considerations for Defining Data Items

In this section we discuss compatibility and usage considerations when defining the type and length of a data item.

Subtopics

1.7.1.4.1 Compatibility Considerations Related to Numeric Data Types

1.7.1.4.2 Using Data Items in SQL Records

1.7.1.4.1 Compatibility Considerations Related to Numeric Data Types

Because of differences in the implementation of numeric data on different execution platforms, you must consider compatibility issues when using numeric data in multiple execution environments. For Bin data types, for example, the data is stored in byte-reversed order in OS/2 compared to the /370 environment. If Bin data is substructured, this may lead to different results on OS/2 than on /370 systems. For Num, Numc, Pack, and Pacf data types, the sign is represented differently in ASCII than in EBCDIC format. This can cause unpredictable results if data of this form is shared between different environments. Refer to the VisualAge Generator manuals and help facility for details regarding compatibility considerations related to numeric data types used on different execution platforms.

1.7.1.4.2 Using Data Items in SQL Records

Table 10 summarizes the compatibility considerations for data items used in SQL records. It shows the subset of possible specifications that can be defined for data items used in SQL records.

Table 10. VisualAge Generator Data Items Used in SQL Records			
Data Type	Digits	Bytes	Decimal Digits
Char	1 to 32767	1 to 32767	N. A.
DBCS	1 to 16383	2, 4, ..., 32766	N. A.
Hex	2, 4, ..., 65534	1 to 32767	N. A.
Pack	1 to 18	1 to 10	0 to 18
Bin	4	2	0
	9	4	0
Notes:			
1. Decimal places are not available for data items within a SQL record if you select a data type other than decimal.			
2. You can specify any number between 1 and 18 as the number of digits for a data item with a data type other than decimal. However, for performance reasons, use only an odd number of digits for packed data items in an SQL record.			

1.7.2 Working Storage Records

When executing a GUI application the user works with different types of data. The data is either entered by the user (for example, the name and address of a new customer), accessed from a database (for example, the vehicles available to be rented), or created by the system (for example, the tax calculated from an amount to be payed). In order to work with data, it has to be stored in memory during the lifetime of the GUI application under execution. In VisualAge Generator the storage used for this purpose is called a *working storage record*. The contents of the working storage record are described in one or more *working storage records* that are defined as nonvisual parts inside a GUI application.

A working storage record itself is a collection of related data items treated as one unit. The data items describe the characteristics of the working storage record data.

Data in working storage record is temporary data, that is, it exists only as long as the GUI application is executed. It is not saved when the GUI application finishes execution. Therefore, if the working storage record data of one GUI application has to be used in another GUI application, it has to be passed to that GUI application.

To define a record you create a new member with type RECD in the MSL. Select **File** and **New member...** from the VisualAge Generator Developer menu and select **Recd** as the type of member to be created. Figure 30 shows the Record Definition window for defining a working storage record.

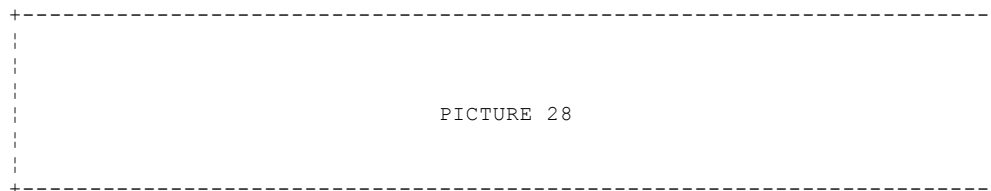


Figure 30. Record Definition Window

On the Record Definition window, you define the following characteristics of a record:

Organization

Looking at the **Organization** group box, you can see that a record can not only be a working storage record but the Record Definition window can also be used to define various other types of data organizations. However, in GUI applications we mainly use working storage records. The other record types are typically used for VisualAge Generator server applications.

Item default scope

If you define new data items from within the Record Definition window, their scope defaults to either *Local* or *Global*, which you select in the **Item Default Scope** group box.

Data items in the record

The **Data Items** list is used to display the data items that make up the record. The data items are shown with all of their properties in the sequence that builds the structure of the working storage record.

GUI PROGRAMMING EXAMPLE: CREATING A WORKING STORAGE RECORD IN THE MSL

Multiple methods are available to open the Record Definition window. In this example the two most commonly used methods for defining working storage records in GUI applications are shown.

1. Using the VisualAge Generator Developer pull-down menu

- a. From the **File** pull-down menu, select **New member**
- b. On the New Member pop-up window click on the **Recd** radio button and the **Open...** button.

The Record Definition - Untitled window is displayed.

Note: The organization defaults to **Indexed** in VisualAge Generator.

- c. Select the **Working storage** radio button to define a working storage record.
 - d. To name the record save it as **RCDTST1**.
2. Using the parts palette of the GUI Application Definition window
- a. Open the GUI Application Definition window for a new GUI application.
 - b. Select the Data Member Parts category and the Record Member Part from the parts palette.
 - c. Move the crosshair to the free-form surface and drop the record member part (mouse button 1).
- The Add Subpart window is displayed.
- d. Enter **RCDTST2** as the name of the new record in the **Add Subpart** drop-down list on the **OK** push button.
- Note:** If you want to use an existing record, click on the drop-down symbol to the right of the records that are yet defined in the MSL and select one of them.
- e. Double-click with mouse button 1 on the record on the free-form surface.
- The Record Definition - RCDTST1 window is displayed as shown in Figure 30. Now the **RCDTST2** member exists in the MSL.
- Note:** The organization defaults to **Working storage** in VisualAge Generator.
- f. Close the GUI application definition window. Do not save the GUI application.

To get the appropriate contents of the pull-down menu and have the left arrow icon displayed on the Developer window, you have to open the Record Definition window. With mouse button 2 drag the arrow icon located to the left of the Developer title bar to the list with data items or choose **Edit** and then **Insert before** or **Insert after** from the menu. In the Record Item Usage window (Figure 31) you define the use of the data item within the record.

PICTURE 29

Figure 31. Record Item Usage Window

On the Record Item Usage window, you define the following properties of a record item:

Name

The name of the data item is mandatory and must follow the VisualAge Generator naming rules. You can use an asterisk (*) instead of a real name for a local data item to represent a *filler data item*. Filler data items represent space holders within the working storage record. You can use Char filler data items with length 1 to get single blanks between adjacent aggregate data items that are to be displayed using a list box instead of a container details view.

A data item other than a filler data item must have a unique name within one working storage record. It can be referenced by this name in processing statements of a logic part of the GUI application in which the working storage record is defined.

Level

Level is a whole number between 3 and 49 or the number 77. It is used to define either data structures (level numbers 3, 4, ..., 49) or single data items (level number 77). The default value is 10. We discuss *Level* in section "Data Structures" in topic 1.7.3.

Occurs

Occurs is a whole number between 1 and 32767. It is used to define data arrays. The default value is 1. We discuss *Occurs* in section "Occurs Items" in topic 1.7.4.

Item Scope

The **Item Scope** is preselected to the value defined in the Record Definition window.

If you specify the name of a global data item that already exists in one of the MSLs of the active MSL concatenation and you select **Global** item scope, the global data item with all of its characteristics is inserted into the record. If you enter a name of an existing global data item and you select **Local** item scope, VisualAge Generator asks you whether you want to use the characteristics of the existing data item as the default for the new local data item.

+-----+ <u>HINT</u> +-----+	
+-----+	
+-----+	
+-----+	
+-----+	
+-----+	
+-----+	
+-----+	
+-----+	
+-----+	

If you click on the **Define...** push button, the (Local) Data Item Definition window is displayed. We describe the Data Item Definition window in section "Defining Data Items" in topic 1.7.1.3.

If you click on the **Insert** push button, the Record Item Usage window closes, and the data item is inserted into the record at the location where you have dragged the object icon. If you want to define more than one data item at the same location in the record, click on the **Next** push button. The data item is inserted, but the Record Item Usage window remains open, and you can define the next data item.

+-----+ <u>HINT</u> +-----+	
+-----+	
+-----+	
+-----+	
+-----+	
+-----+	
+-----+	
+-----+	
+-----+	
+-----+	

After a data item is inserted into the record, its name and all of its characteristics are displayed in the **Data Items** list on the Record Definition Window (see Figure 32). The first untitled column is called the *prefix area*. It shows either the sequence number of the data item within the record, or a dashed arrow if the data item is a global data item that is not completely defined yet.

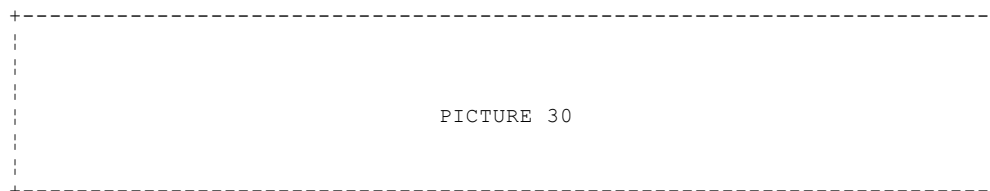


Figure 32. Record Definition Window with Three Data Items Defined

If you move the mouse pointer over the prefix area or the name of a data item, it changes its shape to a Swiss cross to let you know that you can select the area to make updates to the data item. If you select the prefix area, the entire data item is highlighted. If you select the data item name, only the name is highlighted.

If you double-click on the prefix area of a data item with mouse button 1, the Record Item Usage window is displayed. There you can update the usage information for the data item. If you double-click on the name of a data item with mouse button 1, the Data Item Definition window is displayed. There you can update the characteristics of the data item.

For documentation purposes you can define any text to describe the record. To provide a description of the record, select **Prologue** from the **Define** pull-down menu that is active when you are working with the record definition and enter your description.

GUI PROGRAMMING EXAMPLE: DEFINING A WORKING STORAGE RECORD

To define a working storage record, do the following:

1. Open the Record Definition window for the **RCDTST1** record.
2. Select the **Working storage record** and **Local** radio buttons.
3. Drag the object icon to the **Data Items** list and drop it.

The Record Item Usage window for the data item opens. Its scope is preset to **Local**.
4. Enter **ITEM1** as the data item's name and click on the **Next** push button.
5. Enter **ITEM2** as the data item's name and click on the **Next** push button.
6. Enter **ITEM3** as the data item's name and click on the **Insert** push button.

The three items are inserted into the **Data Items** list with sequence numbers 1, 2, 3 and their prefix areas.
7. Double-click on the name of **ITEM1**.

The Local Data Item Definition window opens.
8. Change the length of the data item to 10 bytes and click on the **OK** radio button.
9. Double-click on the name of **ITEM2**.
10. Change the type to **Num** and the length of the data item to six digits with no decimal places and click on the **OK** radio button.
11. Double-click on the name of **ITEM3**.
12. Change the type to **Pack** and the length of the data item to nine digits with two decimal places and click on the **OK** radio button.
13. Maximize the Record Definition window to see all the columns of the data items.

You can see all your definitions and changes reflected in the **Data Items** list. The Record Definition window should look like the one in Figure 32.
14. Close the Record Definition window and save the changes you made to record **RCDTST1**.

Subtopics

1.7.2.1 Accessing Data in Working Storage Record

In this section we describe how data entered in a GUI is moved to a working storage record and how data stored in a working storage record can be displayed on a GUI.

The easiest way to move data entered in a GUI into a working storage record or from a working storage record to a GUI in a GUI application is to define on a window one visual part (for example, an entry field) per single data item of the working storage record and an attribute-to-attribute connection between the data item and the visual part. The connection must connect the `DATAITEM` data attribute of the record to the appropriate attribute of the visual part. For an entry field, for example, this is the `object` attribute. Every time the contents of one of the connected parts are changed, the contents of the other part are changed to the same value as well. Figure 33 shows a GUI application that moves data entered into one text field on the window to the working storage record and back to the other text field.

PICTURE 31

Figure 33. GUI Application That Moves Data between a Window and a Working Storage Record

GUI PROGRAMMING EXAMPLE: MOVING DATA BETWEEN A GUI AND A WORKING STORAGE RECORD

To define a GUI application that moves data from a window to a working storage record, follow the following:

1. Open the GUI Application Definition window for a new GUI application.
2. From the Data Entry category select the Label part.
3. Click the **Sticky** toggle button.
4. Drop two label parts one below the other on the left half of a window part.
They are defined with the default names **Label1** and **Label2** by the GUI builder.
5. From the Data Entry category select the entry field.
6. Drop two entry fields one below the other to the right of the label parts on the window part.
They are defined with the default names **Text1** and **Text2** by the GUI builder.
7. Deselect the **Sticky** toggle button or click on the **Selection Tool** icon on the Tool palette.
8. Align the four parts vertically and horizontally.
9. Open the settings window of the label parts and change their names from **Label1** and **Label2** to **Input** and **Output**.
10. From the Data Member Parts category select the Record Member part.
11. Drop the record part to the right of the window part on the free-form surface.
12. On the Add Subpart window enter **RCDTST2** as the name of the record.
13. Open the Record Definition window and select **Working storage** and **Local**.
14. Define a local data item inside the working storage record with the following characteristics:

Name: ITEM1
Type: Char
Length: 10

Use the default values for all other properties of the data item.
15. Close the Record Definition window and save the record.
16. Open the pop-up menu of the **Text1** part.

17. Select **Connect** and object.

18. Drag the spider to the record **RCDTST2** and drop it. The Connect window for the record opens.

.
.
.

.
.
.

19. Select the attribute ITEM1 data and click on the **OK** push button.

This defines an attribute-to-attribute connection between the two parts.

20. Define an attribute-to-attribute connection of the same kind between **Text2** and the data attribute of **RCDTST2**.

21. Test the GUI application (save it as member **GUITST3**).

22. Enter data in the **Input** field.

The data entered is displayed in the **Output** field. This proves that the data is transferred from the **Input** field to the record and from the record to the **Output** field.

23. Now enter data into the **Output** field.

The data entered is displayed in the **Input** field. The reason for this is that an attribute-to-attribute connection between two parts is symmetric, that is, if the contents of one part change the contents of the other part are updated automatically.

Note: If you add more than three characters to either the **Input** or **Output** field, the data is not displayed in the other field because the **ITEM1** data item has a length of three characters.

1.7.2.2 Quick Form

Quick Form is a convenient way of creating visual parts from the definitions of most VisualAge Generator parts. Quick Form enables you to choose an attribute of the source part that you want to have displayed. Quick Form creates for the attribute:

- A Label part with the description (or name) of the attribute. If a description was defined for the data item, this text is used as the label text. If a description was not defined, the name of the data item becomes the text of the label.
- An appropriate visual part according to the type of the attribute
- A data type and character limit specification for the visual part that fits the characteristics of the source part
- A connection between the attribute of the source part and the appropriate attribute of the created visual part. The connection keeps the contents of the two parts synchronized. The kind of connection depends on the type of the source and visual parts.

If you use the self attribute of the source, part Quick Form creates the visual parts and connections for every attribute of the source part. Figure 34 shows a GUI application definition with visual parts created from a working storage record record by using Quick Form and the self attribute.

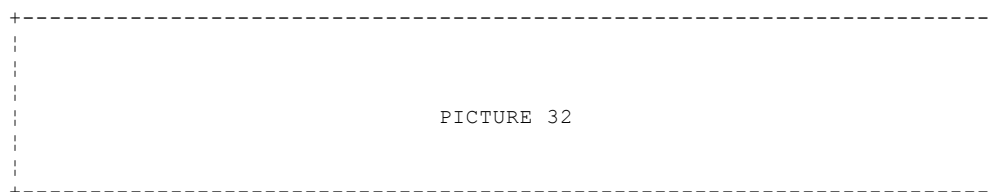


Figure 34. Visual Parts Created with Quick Form

Quick Form is especially useful for creating visual parts on a window from data items of a working storage record that are used to enter data that is moved to working storage record and for displaying data stored in a working storage record. Instead of creating the visual parts manually, you can use Quick Form to automatically define these parts together with a descriptive label for each.

GUI PROGRAMMING EXAMPLE: USING QUICK FORM	
	To use Quick Form to create the appropriate definitions of visual parts from the data record part, do the following:
	1. Open a new GUI application. Define record RCDTST1 on the free-form surface of the application. You created record RCDTST1 in the Defining a Working Storage Record topic 1.7.2.
	2. Open the pop-up menu of RCDTST1 and select Quick Form . A selection list is displayed that contains the names of the data items of the record and <u>self</u> as attributes of the
	.
	.
	.
	3. Select <u>ITEM1</u> , move the mouse pointer in form of a cross-hair near the upper-left inside the window and click with mouse button 1.
	The GUI builder creates a label, entry field, and connection to the <u>ITEM1</u> data item. This is the same kind of connection we defined manually in the Moving Data between a GUI and a Working Storage Record in topic 1.7.2.1.
	4. Open the pop-up menu of RCDTST1 and select Quick Form again. Select the <u>self</u> attribute, move the mouse pointer below the <u>ITEM1</u> label, and click with mouse button 1.

The GUI builder creates a label, entry field, and connection for each of the data items in the record.

5. Open the settings of the entry fields and look at the Input Field Specifications window of the Quick Form.

You see that the data type and the character limit of the text field were set according to the data type and length of the data item.

6. Open the Record Definition window of the record.
7. Open the Data Item Definition window for each data item (click on its name).
8. Define descriptions for each data item as follows:

```
ITEM1: Cust-Name
ITEM2: Cust-Number
ITEM3: Amount-to-pay
```

9. Close the Record Definition window and save the changes.

Note: The label text in the window part in the GUI application did not change.

10. Delete all the parts inside the window of the GUI application.
11. Repeat the definition of visual parts from record **RCDTST1** using Quick Form and save the changes.

Now the descriptions of the data items are used as labels for the text fields.

12. Test the GUI application (save the member as **GUITST4**).

Enter data in the entry fields. Select **Options** and then **Application data...** from the Test Monitor window. Select **GUITST4** from the top list and then **RCDTST1** from the bottom list. Click on the **View...** push button and see whether the data is now also part of the working storage record.

Note: If you get the message *****error***** you triggered a validation error. This is discussed in "Data Validation Errors" in topic 2.3.2.

1.7.3 Data Structures

A data structure is an agglomeration of data items generally with differing characteristics defined within a record or table. For example, a record with more than one data item defined is a data structure by itself. A *(sub)structured data item* or *structure* is an aggregation of other data items, called *substructure data items* or *substructures*. The substructures can be either elementary data items or structured themselves. Filler data items can be structures or substructures too.

Subtopics

1.7.3.1 Using Data Structures

1.7.3.2 Accessing Data in Data Structures

1.7.3.1 Using Data Structures

Data structures can be used for various reasons. One reason for using data structures is that they enable you to define overlays of single data items with different data types. For example you can define a character data item that is substructured with a numeric data item of the same length. The two data items overlay each other and therefore describe the same data. If you know the type of data in the data items, you can use either the character or the numeric data item in appropriate processing statements of a logic part without having to have VisualAge Generator convert the data from one format to the other.

Note: There are disadvantages and possible runtime errors that can be caused when data types are overlapped in substructured data items. There are generation options (**/SPZERO** and **/CHETYPE**) that can be used to alter the way numeric data types are evaluated and check for possible data type conflicts in substructured data items. See the *Generating VisualAge Generator Applications* for more details.

You can also subdivide a data item into several other definitions. For example, by substructuring a credit card number, you can access the valid-through date without having to substring the data.

A structure makes up a hierarchy of data definitions, allowing you to access individual elements in the hierarchy.

To define a structured data item in a record or table you use the relative *level* of those data items that are part of the structure. The rules for *level* are as follows:

- ☐ The lower the level of a data item, the higher it is in the hierarchy of the data structure of the record.
- ☐ The data item or items with the lowest level number represent the top data elements within the record structure.
- ☐ A data item with a higher level number is a substructure of the previous data item in the structure with a lower level number.
- ☐ Substructures of a structure have the same level number.
- ☐ A structured data item is the aggregation of its substructures with the next higher level number in the sequence of their positions.
- ☐ The sum of the byte lengths of the substructures of one structure must be equal to the length of the structure.
- ☐ Level information is unique to a record or table. It can be different for the same (global) data item used in other records or tables.

When you create a record containing structured data, you have to calculate and define the correct length of all structured data items. However, you can let the VisualAge Generator Developer record definition editor either calculate the length of a data item or validate the data structure of the whole record. The validation checks for each structured data item if its length specified is the sum of the lengths of its substructures. In addition it checks whether the names of all data items of the record are unique.

HINT

To let the record definition editor calculate the length of a data item, select the row in the **Data Items** list of the Record Definition window and select **Calculate substructure length** from the **Options** pull-down menu.

You can ask the record definition editor to validate the data structures of the record selecting **Validate data structure** from the **Options** pull-down menu. Validation is always performed when you save the member.

GUI PROGRAMMING EXAMPLE: DEFINING A STRUCTURED WORKING STORAGE RECORD

To create a structured working storage record, do the following:

1. Open a new GUI application.
2. Define a working storage record **RCDTST3** on the free-form surface.
3. Open the Record Definition window for the record **RCDTST3**.
4. Select **Local** as **Item Default Scope**
5. Define the data items shown below within **RCDTST3**.

NAME	SCOPE	LEVEL	OCCURS	TYPE	LENGTH	DECIMAL	DESCRIPTION
CNAME	Local	10	1	Char	25	N.A.	Name of Client
DTE	Local	10	1	Char	3	N.A.	Date of Birth
YEAR	Local	15	1	Char	2	N.A.	Year of Birth
MON	Local	15	1	Char	2	N.A.	Month of Birth
DAY	Local	15	1	Char	2	N.A.	Day of Birth
ADDR	Local	10	1	Char	3	N.A.	Address
NR	Local	15	1	Num	5	0	Number
STR	Local	15	1	Char	20	N.A.	Street
TOWN	Local	15	1	Char	20	N.A.	Town
ZIP	Local	15	1	Char	3	N.A.	Zip Code
STATE	Local	20	1	Char	2	N.A.	State
PC	Local	20	1	Num	6	0	Post Code
PR	Local	10	1	Pack	9	2	Premium

6. Select the ADDR data item in the Record Definition window (click on its prefix and then on its data item name).
7. Select **Calculate substructure length** from the **Options** pull-down menu.

An Information window pops up telling you that the total length of the substructure item ADDR is 53 bytes.
8. Change the length of data item ADDR to 53 bytes. To open the data item editor from the Record Definition window, double-click directly on the data item name.
9. Select **Validate data structure** from the **Options** pull-down menu to check whether the values of the structures are correctly defined. If necessary, correct them.
10. Close the Record Definition window and save your changes made to the record.
11. Save the GUI application as **GUITST5** for later use.

The data items with level numbers between 3 and 49 define the data structure of a record. Instead of, or in addition to, the data structure, you can define one or more single, unrelated data items within a working storage record. Single data items are referred to as level-77 data items. They are defined with a level of 77 and have to be defined as the last data items of the working storage record. Level-77 data items can neither be structured nor represent an array. Use level-77 data items to store intermediate data, such as counters, indexes, or intermediate results of calculations, created for temporary use in a GUI application. They can also be good choices for data triggers (see "Data Item" in topic 1.9.2.2) and for connecting to fields in a GUI application.

Because they are not part of the data structure of the working storage record VisualAge Generator treats level-77 data items differently from non-level-77 data items in the following circumstances:

- ☐ If the working storage record is changed, the level-77 data items are not triggered.
- ☐ If the working storage record is passed as a parameter in a CALL statement to a server application, the level-77 data items are not passed with the record.

Consider these differences in the treatment of working storage record data items when choosing between level-77 or non-level-77 data items at definition time. The use of level-77 data items will reduce unnecessary triggering.

1.7.3.2 Accessing Data in Data Structures

If you put a working storage record with a data structure in a GUI application, you will notice that the Connect window of the working storage record contains only those data items with the lowest level. The substructure parts are not visible. They are contained within the data items with the lowest level indicator. So how can these substructures be accessed?

Any attribute of any part within VisualAge Generator can be *torn off*. Tearing off an attribute exposes all the attributes, actions, and events available to that attribute. In the case of a data item, this includes its substructured items, which are its attributes.

To tear off an attribute of a part, select **Tear-off Attribute...** from the Object menu and select the attribute to tear off from the list of attributes that is presented to you. If there is text in brackets to the right of the attribute, it tells you what type the attribute is. For example, *Compound* in brackets informs you that the attribute at its left represents a structured data item.

A variable part representing the attribute that was torn off is placed on the free-form surface. The part is called **attribute of partName**. The torn-off attribute is connected to the part from which it was torn off by an attribute-to-attribute connection connecting the attribute to the self attribute of the variable part. You can now perform actions on the torn-off attribute, for example, you can create a Quick Form from one of its attributes. If you use Quick Form on self of the torn-off record attribute, you create parts for each of the substructured data items. Figure 35 shows an example of torn-off attributes in a GUI application.

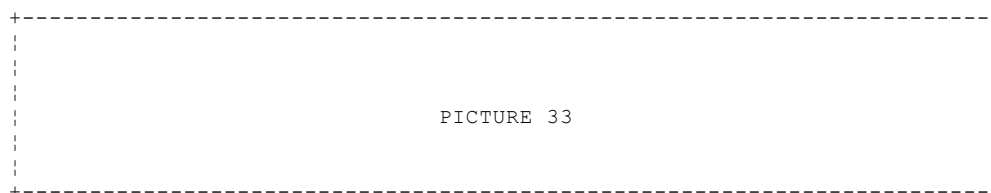


Figure 35. Visual Parts Created from a Structured Working Storage Record Using Tear-Off and Quick Form

GUI PROGRAMMING EXAMPLE: TEAR-OFF ATTRIBUTES

To tear-off attributes of a working storage record part, do the following:

1. Open GUI application **GUIST5**.
2. Resize the window part so that it is longer (see Figure 35).
3. At the top of the window define the visual part for CNAME and at the bottom of the window define the visual part for PR from record **RCDTST3** using Quick Form.

You can achieve the same result using Quick Form and self and delete all labels and fields of the parts not needed.
4. From the pop-up window of record **RCDTST3** select **Tear-Off Attribute...**
5. Select **DTE (Compound)** from the selection list.
6. Drop the hair-crossed message prompter on the free-form surface.

The GUI builder creates a variable part representing a data item called **DTE of RO**

7. Using Quick Form and the self attribute of the torn-off attribute **DTE of RCDTST3**, visual parts for DAY, MON, and YEAR.
8. Tear-off the **ADDR** data item from **RCDTST3**.

The GUI builder creates a variable part representing a data item called **ADDR of 1**

9. From **ADDR of RCDTST3** define the visual parts for all nonstructured data items on using Quick Form (these are all data items except **ZIP**).
10. Tear-off the **ZIP** data item from **ADDR of RCDTST3**.

The GUI builder creates a variable part representing a data item called **ZIP of ADDR of RCDTST3**.

11. From **ZIP of ADDR of RCDTST3** define the visual parts for STATE and PC, using Quick
12. Save the GUI application **GUITST5** in your MSL.

1.7.4 Occurs Items

VisualAge Generator also allows you to define arrays of data in a working storage record. You can work with one individual element of an array, using its position number within the array, called its *index*; access a group of array elements that occupy adjacent positions, using the start and end index of the group; or work with the whole array as a part on its own.

Subtopics

1.7.4.1 Using Occurs Items

1.7.4.2 Accessing Data in Occurs Items

1.7.4.1 Using Occurs Items

To define a data array in a working storage record, you use the *Occurs* attribute of a data item that represents the array. The characteristics (data type, length, decimal places, description) of that data item describe the characteristics of all array elements. The Occurs attribute defines the number of occurrences of the data item within the record. The occurs characteristic for a data item applies only for the record where the characteristic is defined. The number of occurs cannot be changed dynamically.

Data items with an occurs value greater than 1 are called *occurs items*. An occurs item can be substructured, but none of the data items within its substructure can be defined with an occurs value greater than 1. Level-77 data items cannot have an occurs value greater than 1. Figure 36 shows a working storage record with various forms of aggregate data.

PICTURE 34

Figure 36. Working Storage Record Representing Various Forms of Aggregate Data

GUI PROGRAMMING EXAMPLE: DEFINING OCCURS ITEMS

To add simple and substructured occurs items to an existing working storage record you do the following:

1. Open GUI application **GUITST5**.
2. Open the Record Definition window for **RCDTST3**.
3. Define the data items shown below at the end of the data structure of the record:

NAME	SCOPE	LEVEL	OCCURS	TYPE	LENGTH	DECIMAL	DESCRIPTION
COND	Local	10	3	Num	3	0	Conditions
MCONS	Local	10	12	Char	16	N.A.	Monthly Consumption
GAS	Local	15	1	Pack	9	2	Gas (cbm)
ELEC	Local	15	1	Pack	9	2	Electricity (kWh)
WATER	Local	15	1	Pack	7	4	Water (cbm)
TEL	Local	15	1	Bin	4	0	Telephone (Impulses)

4. Close the Record Definition window and save the changes.

Although an occurring structure cannot contain another occurring structure, you can implement multiple dimensional arrays in VisualAge Generator by combining the characteristics of structured and occurring data items. You can define a working storage record with a data item that occurs multiple times. The contents of one of the elements of this array as identified by its index can be moved to another working storage that contains a substructure that also contains an occurring structure. Each element in the array is thereby defined as being an array itself.

GUI PROGRAMMING EXAMPLE: DEFINING MULTIDIMENSIONAL ARRAY

To create a working storage record structure that allows you to store multiple messages, where each message in turn contains multiple placeholder values, do the following:

1. Create a new working storage record.
2. Define the data items shown below:

NAME	SCOPE	LEVEL	OCCURS	TYPE	LENGTH	DECIMAL	DESCRIPTION
MESSAGE	Local	10	10	Char	103	N.A.	Messages
MSGCODE	Local	12	1	Char	3	N.A.	Message Code
VALUE	Local	12	1	Char	100	N.A.	Placeholder Values

3. Save the working storage record as **MSGRCD1**.

4. Create a second working storage record.

5. Define the data item shown below:

NAME	SCOPE	LEVEL	OCCURS	TYPE	LENGTH	DECIMAL	DESCRIPTION
VALUE	Local	10	4	Char	25	N.A.	Placeholder Value

6. Save the working storage record as **MSGRCD2**.

7. By moving data from a certain index of the array of **VALUE** in the first working storage record to **VALUE** in the second working storage record, you would be able to access placeholder values.

1.7.4.2 Accessing Data in Occurs Items

An occurs item, when torn off from the working storage record, has a number of actions that allow you to access the individual elements in the array:

getFieldAtIndex:

Gets a pointer to the element at the index that is passed as the parameter to the connection

getValueAtIndex:

Gets the value of the element at the index that is passed as the parameter to the connection

setValueAtIndex:Using:

Changes the value at the specified index, the anIndex attribute of the connection, to the value specified in the value attribute of the connection

getFieldsStartingAt:to:

Gets a pointer to a range of elements starting at the index that is passed as the firstIndex parameter and ending at the index that is passed as the lastIndex parameter. This action can be used to improve the performance of loading table and container details parts with array data.

getValuesStartingAt:to:

Gets the values of a range of elements starting at the index that is passed as the firstIndex parameter and ending at the index that is passed as the lastIndex parameter. This action can be useful for loading lists, combos, and radio button sets with array data.

setValuesStartingAt:to:using:

Changes the values starting at the specified index, the firstIndex attribute of the connection, until the second specified index, lastIndex, to the values specified in the aCollection attribute of the connection

You should always use these actions when accessing an array of data from a working storage record because it requires you to specify a specific element or range of elements in the array to access. This ensures that VisualAge Generator does not have to figure out where the array no longer contains relevant data (blanks or zeros), which is a very expensive bit of processing, especially for sparsely filled arrays.

When you use the Quick Form facility of the GUI builder to create a representation of an array, it creates an attribute-to-attribute connection. You should always delete that connection and replace it with a connection, using one of the actions for accessing array elements.

GUI PROGRAMMING EXAMPLE: USING OCCURS ITEMS

To access and use an occurs item in your application, do the following:

1. Open GUI application **GUITST5**.
2. Tear off the MCONS attribute of the **RCDTST3** working storage record.
3. Select **Quick Form** from the pop-up menu of the MCONS torn-off attribute.
4. Select **self** from the selection list and click on the window part of the GUI application. A label, a container details, and an attribute-to-attribute connection are created.
5. Click on the attribute-to-attribute connection to see the attributes involved in the connection.

The connection connects the MCONS attribute of the record to the items attribute of the Container Details view.
6. By default the Quick Form does not create a connection using the getFieldsStartingAt:to: action on the occurring data item. Delete the current connection to the container details view.
7. Determine which event should refresh the data in the list, for instance, aboutToClose.

event of the window part. Connect this event to the getFieldsStartingAt:to: action of the MCONS part. Connect the torn-off attribute MCONS.

8. Open the settings of the connection and click on the **Set parameters...** push button. Set the first index to 1 as the first index and 12 as the last index. Normally you would provide these values dynamically. Click on **OK** to close the window and on **OK** again to close the settings window.

9. Connect the result attribute of the connection (this contains the pointer to the container details) to the items attribute of the container details. To connect to the container details column of the container details, click on the scroll bar for the part.

10. Save the GUI application as **GUITST6**.

1.7.5 VisualAge Generator Tables

A working storage record loses its contents when you close the application. You also have to pass the data between different GUI applications for them to be able to share the data. The VisualAge Generator table provides a solution for those situations where you have data that should be accessible to all or most applications, where the contents are relatively stable, and you can break the atomic nature of the code.

Subtopics

- 1.7.5.1 VisualAge Generator Table Characteristics
- 1.7.5.2 Using a VisualAge Generator Table

1.7.5.1 VisualAge Generator Table Characteristics

A VisualAge Generator table is, in structure, identical to an occurring structure in a working storage record. It contains as many rows as you define it to have and each row can be substructured. In contrast to a working storage record, a VisualAge Generator table does not contain a specification of the number of items it can contain. Rather it has an initial contents and can never contain more rows than were initially defined for the table (although these rows can be empty).

GUI PROGRAMMING EXAMPLE: DEFINING A VISUALAGE GENERATOR TABLE	
	<p>To define a VisualAge Generator table and provide it with initial contents, do the following:</p> <ol style="list-style-type: none"> 1. Create a VisualAge Generator table member using the New member... option from the menu. Double-click on the table icon. 2. Set the type of the table to Unspecified. The other table types require a specification but are only used by VisualAge Generator TUI applications. 3. Set the Item Default Scope to Local. 4. Make sure Fold table contents is not checked. If it is, all the contents of the VisualAge Generator table will be converted to upper case. 5. Set the table to resident and shared. If the VisualAge Generator table is not shared, each VisualAge Generator table in each GUI application has its own independent data. A shared VisualAge Generator table is automatically shared. A VisualAge Generator table that is not shared is released from memory when the VisualAge Generator GUI application runtime services are killed. 6. Define the following structure and data items to be used in the VisualAge Generator table: <pre> 10 FIRSTITEM char 5 10 SECONDITEM char 5 </pre> <p>Use the Object icon (the arrow in the top-left corner of the VisualAge Generator table icon) to add data items to the table.</p> <p>Notice that you cannot set the occurs of the data item or make a level-77 data item in the VisualAge Generator table. You can substructure items, however.</p> 7. Select Define and then Table contents... from the menu. You will get a spreadsheet form where you can define the initial contents of the VisualAge Generator table. 8. Add a number of rows of data. <p>Use the Object icon to add entries in the list of table contents. Click on OK and the VisualAge Generator table as member VGTBLE.</p>

The initial contents of the VisualAge Generator table have to be stored somewhere. They are stored in a file. In fact a VisualAge Generator table is generated as a separate file in your final runtime application. It is generated as a text file containing the initial entries in the table. It is initially loaded into memory and kept in memory as long as the VisualAge Generator runtime services are running.

Although the VisualAge Generator table has initial contents, its contents can be changed during execution of the application. The data can be changed by using procedural code or visual logic. The contents of the VisualAge Generator table are shared across GUI applications if you have defined the VisualAge Generator table to be shared. Thus any changes made to the VisualAge Generator table in one GUI application are automatically picked up by the VisualAge Generator table in another GUI application, without having to have any attribute-to-attribute connections between the GUI applications. If you use the same VisualAge Generator table in a server application, the changes are not picked up automatically.

Although one or more attributes change when you change the contents of a VisualAge Generator table, the events related to these attributes do not get signaled. You have to explicitly refresh or access the data again to get the new values.

GUI PROGRAMMING EXAMPLE: USING A VISUALAGE GENERATOR TABLE

To share data in the VisualAge Generator table between GUI applications, do the following:

1. Create a GUI application and drop a VisualAge Generator table part on the free-form surface. Choose the VisualAge Generator table previously defined from the drop-down list (see Figure 1).
2. Create a quick form of the VisualAge Generator table table columns attribute on the free-form surface. This creates a container details, the default visual part for lists of data. Arrange the container details columns to improve the viewable area.
3. Save this GUI application as member **GUI TAB2**.
4. Save this GUI application again but this time as member **GUI TAB1**. We are not done with the GUI application, so do not close the GUI builder window.
5. Add a push button to the window of the GUI application. Add **GUI TAB2** as an external application to the free-form surface of **GUI TAB1**. Connect the clicked event of the push button to the openWidget action of the external GUI application.

.

.

6. Save **GUI TAB1** again.
7. Test the application. Open the second application by clicking on the push button. Change data in either of the two container details.

Experiment to find methods of triggering *visibility* to the changed VisualAge Generator table data in the two different GUI applications. These techniques work:
 - a. Close the **GUI TAB2** GUI application and reopen it using the push button in the first GUI application.
 - b. Edit a cell in one GUI application. Click on that same cell, and then another cell in the other GUI application on the other GUI application.
8. Make these changes to the **GUI TAB2** GUI application to control when the data is refreshed:
 - a. Add a push button to the window in **GUI TAB2**.
 - b. Connect the clicked event of the push button to the refreshAllItems action of the container details in the window.
9. Save **GUI TAB2**.
10. Test the **GUI TAB1** application. Open the second GUI application by clicking on the push button. Change data in the container details of the first GUI application.

Now when you click on the push button in the second GUI application, the container details data will be refreshed from the source, the VisualAge Generator table.

A VisualAge Generator table has a number of attributes that are different from those used by a working storage record. The table columns and table columns data attributes are comparable to the occurring data item attributes of a working storage record.

HINT

The contents of a VisualAge Generator table, when modified during a test cycle, remain in memory. If you want to refresh the contents of the VisualAge Generator table from the definition, stop the ITF. When you next start a test run, the VisualAge Generator table contents will match the contents defined for the member.

--	--

1.7.5.2 Using a VisualAge Generator Table

The concept of the VisualAge Generator table in a GUI application is very powerful. It enables you to use global variables. But just as in any other programming language, the use of global variables abuses the modularity and encapsulation of your applications. You should be well aware of the purpose for which you use the VisualAge Generator table in your GUI applications.

A number of possible uses for the VisualAge Generator table are:

- ☐ Reference tables (such as a table of airports and airport codes) with data that does not change often and could be fetched from a database when first accessed.
- ☐ System-related tables, such as tables with message codes and security information. If you define the data statically, that is, as initial contents, you have to be aware that the data is ultimately stored as a file. The file is editable and should therefore be placed on a secured LAN drive.
- ☐ System-related information that would otherwise have to be passed along to all applications (true global data).

1.7.6 Ordered Collections

In addition to providing working storage records and VisualAge Generator tables to store data, VisualAge Generator offers the *ordered collection*, a dynamic array that can store any type of data and does not have a size limit. In this section we discuss the characteristics of the ordered collection, how it can be used, and its relationship to a working storage record and the VisualAge Generator table.

Subtopics

1.7.6.1 Characteristics

1.7.6.2 Use

1.7.6.3 Ordered Collection as a Tear-Off

1.7.6.1 Characteristics

The ordered collection is a part in the parts palette of the GUI Application Definition window. It is part of the Models category. The ordered collection is a nonvisual part.

An ordered collection is an array with an unlimited amount of elements. Each element of the array can contain any type of information which can differ from the other elements in the array. The data type stored at a certain element in the array does not have to be defined. The action using the information from the array is responsible for understanding its type. For example, an ordered collection can contain other ordered collections.

The size of the ordered collection is dynamic. Adding an element to the array increases its size, and removing an element decreases its size. Each element of the ordered collection can be referenced by its index. An entry in the array does not have to be unique.

The elements of an ordered collection cannot be accessed in procedural logic. To access the data in an ordered collection, you always have to move the data to working storage record or a VisualAge Generator table.

1.7.6.2 Use

The ordered collection can be accessed and used like any other part. An ordered collection has the following actions:

Adding a single item to the collection

Use add:, addAfterIndex:, or addBeforeIndex:, depending on where you want to add the element. The add action adds the item to the end of the list.

Adding a collection of items

Allows you to add a collection of items such as an occurring structure of a working storage record to the ordered collection. Use addAll:, addAllAfterIndex:, or addAllBeforeIndex:, depending on where you want to add the collection.

Modifying a single item in the collection

The at:put: action will replace the value at the specified index.

Removing one or more items

Use remove: to remove a specific item. Provide the item to remove as a parameter. Use removeAtIndex: to remove an item at a certain index. Provide the index as a parameter. Use removeAll to remove a collection from the ordered collection. Provide the collection as a parameter. Provide the self attribute of the ordered collection as a parameter to make the ordered collection empty.

Extracting or one or more items

One or more items can be extracted from an ordered collection by using the copyFrom:to: action. The From: and to: actions represent the indices of the range required. The result of the action is an ordered collection of the elements in the specified range.

Finding items

Items can be located in an ordered collection by providing the index: at: and atIndex:. The index has to be provided as a parameter. An invalid index causes a run-time error. The actions place the value in the result attribute of the connection. Items can also be located by their value: includes: and indexOf:. The former returns a Boolean indicating whether the value was found or not. The latter returns the index in the result attribute of the connection. The first and last item in the ordered collection are always present as attributes of the ordered collection: first and last, respectively.

Size information

The isEmpty attribute indicates whether the ordered collection is empty. It contains a Boolean value. The size attribute of the ordered collection indicates the number of elements in the ordered collection. If isEmpty is true, size is zero.

The ordered collection also has an attribute called iterator. This attribute is explained in more detail in "Iterations" in topic 1.9.3.

GUI PROGRAMMING EXAMPLE: USING ORDERED COLLECTIONS

To add elements to an ordered collection and remove them, do the following:

1. Create a new GUI application.
2. Put a list box, three push buttons, and an entry field on the window. Give the labels "Add," "Delete," and "Clear."
3. Add an ordered collection to the free-form surface of the GUI application.
4. If you click on the **Add** push button, you add the entry in the entry field to the collection. Connect the clicked event of the **Add** push button to the add: action ordered collection. Provide the object attribute of the entry field as the parameter (anObject attribute) to this connection.
5. To show the ordered collection in the list box connect the self attribute of the collection to the items attribute of the list box.
6. If you click on the **Delete** push button, you will remove the selected entry from the box. Connect the clicked event of the **Delete** push button to the remove: action of the collection.

ordered collection. Provide the selectedItem attribute of the list box as the parameter for the connection.

7. To clear the list connect the clicked event of the **Clear** push button to the remove action of the ordered collection. Connect the self attribute of the ordered collection to the parameter (aCollection) to this connection.
8. Test the GUI application (save the member as **GUITST7**).

Figure 37 shows the result of this example.

PICTURE 35

Figure 37. Using the Ordered Collection

1.7.6.3 Ordered Collection as a Tear-Off

Attributes can be torn off. A number of attributes of parts occur multiple times, such as an occurring structure in a working storage record and the `items` attribute of a list box. Tearing these attributes off gives you a torn-off attribute that looks just like an ordered collection.

This characteristic allows you to use all of the features of an ordered collection to find, add, and remove data in working storage record, a VisualAge Generator table and any other attribute that contains multiple values. If you try to add more items to the tear-off than allowed by the VisualAge Generator table or working storage record, you get a run-time error.

The actions of the ordered collection allow you to remove entries from a working storage record or VisualAge Generator table. When the entry is removed, the remaining entries shift in the array to fill in the gap left by the entry that was removed. This is done with the ordered collection action instead of procedural logic.

GUI PROGRAMMING EXAMPLE: TEARING OFF ORDERED COLLECTIONS

To use a torn-off ordered collection to change the contents of a VisualAge Generator table, follow the following:

1. Create a new GUI application. Put a label and push button on the window.
2. Put a VisualAge Generator table on the free-form surface. On the Add Subpart window, enter **VGTABLE** as the name of the VisualAge Generator table. This was created in Defining VisualAge Generator Table in topic 1.7.5.1.
3. Make a quick form of the VisualAge Generator table attribute `table columns` and drag it to the window. This will show a container details with the data.
4. Tear-off `table columns`. From this torn-off attribute, tear off the `data` attribute.
5. Connect the `size` attribute of this ordered collection to the `object` attribute of the window.
6. Connect the `clicked` event of the push button to the `removeAtIndex:` action of the ordered collection. Provide 1 as a hard coded parameter to the connection.
7. Test the GUI application (save it as member **GUITST8**).

Every time you click the push button, the first entry is removed from the VisualAge Generator table and the size decreases by 1. Go to the Test Monitor window. Select **Application Data...** from the menu. Select the name of your GUI application in the top list and the name of your VisualAge Generator table in the bottom list. Click the **View...** push button. Select one of the VisualAge Generator table data items and click the **Occurs...** push button. This will open an Occurs Data View window.

Notice that the VisualAge Generator table still has the number of occurs that were originally added to it. Now the last data item instances have been filled with blank.

GUI PROGRAMMING EXAMPLE: TEAR OFF ORDERED COLLECTIONS

Torn-off attributes of other parts can also be valuable. This example shows the usage of `selectedItems`.

This example shows the possibility of the tear-off ordered collection to change the contents of a VisualAge Generator table. The example builds on the previous GUI programming example, Ordered Collections in topic 1.7.6.2, GUI application **GUITST7**.

1. Replace the list box by a multiple select list.

Move the connection from the ordered collection `self` attribute to the multiple select list `items` attribute. (You can do this by first adding the multiple select list and then deleting the connection from the list box to the multiple select list. Delete the list box window.)
2. Delete the connection from the **Delete** push button to the ordered collection.

3. Put two extra push buttons on the window. Give them the labels "Select all" and "Deselect all."
 4. Tear off the selectedItems attribute of the multiple select list. Notice that in the tear-off selection list the selectedItems attribute is identified as an ordered collection. Put parentheses behind the attribute name.
 5. Connect the clicked event of the **Select all** push button to the addAll: action of the selectedItems attribute. Provide the self of the ordered collection that contains the selectedItems attribute as the parameter to the connection.
 6. Connect the clicked event of the **Deselect all** push button to the removeAll: action of the selectedItems attribute. Provide the self of the selectedItems attribute as the parameter to the connection.
 7. Delete the **Clear** push button. Change the connection from the **Delete** push button. Instead of connecting to the remove: action, connect it to the removeAll: action and provide the self attribute of the selectedItems attribute as the parameter to the connection.
 8. Save the GUI application as member **GUITST9**.
 9. Test the GUI application.
- Check whether the selected entries are deleted from the list if when you click **Deselect all**. Check whether you can select and deselect all the values in the list.

1.7.7 *Sharing Data*

Data is often not restricted to one GUI application but has to be shared among different GUI applications. VisualAge Generator provides a number of mechanisms for sharing data. In this section we describe those mechanisms.

Subtopics

1.7.7.1 Promoting Features

1.7.7.2 Passing Data

1.7.7.3 Global Variables

1.7.7.1 Promoting Features

The basis for sharing data is the ability to add attributes as a feature of a GUI application, thereby enabling you to access the data like any other attribute of the GUI application. To add a feature to the interface of a GUI application, choose **Promote part feature...** from the Object menu of the part for which you want to make an attribute available in the interface of the GUI application. Figure 38 shows the window from which features are promoted to the interface of the embeddable part.

PICTURE 36

Figure 38. Window for Promoting Features to the Interface

Select the action, attribute, or event from the list and provide it with a meaningful name (see Appendix B, "VisualAge Generator Naming Convention" in topic B.0 for more information about naming promoted features). Click on the **Promote** push button to add the feature to the list of promoted features.

To remove a promoted feature, select it from the list of promoted features of a part and click on the **Remove** push button. To rename a promoted feature, you have to remove it and then promote it again.

To see all of the features of the GUI application, including your newly added feature, bring up the Connect window of the free-form surface.

GUI PROGRAMMING EXAMPLE: PROMOTING A FEATURE

To add a feature to the interface of a GUI application and use it to share data with application, do the following:

1. Create a new GUI application. Put an entry field on the GUI application.
2. Select **Promote Part Feature...** from the Object menu of the entry field.
3. From the list of attributes select object. Replace the default name in the **Promoted name** field with "customerNameData."
4. Click on **Promote**. The feature is added to the list of previously promoted features in the window by double-clicking on the system menu.
5. Save the GUI application as member **GUISHR1**.
6. Create a new GUI application. Add the previously created GUI application by dropping application member part on the free-form surface and selecting the GUI application from the list.
7. Put an entry field and a push button on the window. Connect the clicked event of the button to the openWidget action of the external GUI application **GUISHR1**.
8. Connect the object attribute of the entry field to the customerNameData attribute of the external GUI application.
9. Test the GUI application (save it as member **GUISHR2**).

Change the name in either of the entry fields and notice that the value in the other field also changes. Now make the attribute-to-attribute connection unidirectional and test the application again.

Sharing data between GUI applications requires that you promote the data you want to share. The attribute we promoted in the previous GUI programming example is the object attribute of an entry field. We could have used any two attributes that have a compatible type, for example, data items from a working storage record.

1.7.7.2 *Passing Data*

In procedural programming languages, data can be shared between a program and a function it calls by either passing the value or by passing a pointer to the value. In the first case, the function gets its own copy of the data. In the second case, the function works on the same copy of the data the program used.

When making a connection between two attributes of two GUI applications (such as object of an entry field or DATAITEM data of a working storage record), you are in fact neither passing by value nor passing. Instead two copies of the data are continuously kept synchronized. This can cause a lot of events, especially when more than two GUI applications are involved. In most cases you should avoid attribute-to-attribute connections between GUI applications. Instead use pass by value or pass by reference implementations.

Subtopics

1.7.7.2.1 Pass by Value

1.7.7.2.2 Pass by Reference

1.7.7.2.1 Pass by Value

With pass by value, a copy of the data is passed when it is needed by the GUI application. Instead of using an attribute-to-attribute connection, you therefore use an event-to-attribute connection and provide the value of the attribute that has to be changed as the parameter.

If you change the value in the GUI application to which it is passed, the change does not effect the original data that represents the source. If you want to provide the ability to pass back the changed value as well, you would have to have the GUI application that changed the value signal an event indicating that it has finished executing the function. You can promote an event in the GUI application for this purpose. Use this event, in the related GUI application, as the start of an event-to-attribute connection, to update the source value. Provide the promoted attribute that was used to pass the value to the GUI application that changed it, as the parameter for the update of the source of the value.

GUI PROGRAMMING EXAMPLE: PASS BY VALUE

This example builds on the Promoting a Feature in topic 1.7.7.1. To pass data using value technique do the following:

1. Open GUI application **GUISHR2**.
2. Remove the attribute-to-attribute connection.
3. Make a second connection from the push button. Connect the clicked event of the to the customerNameData attribute of the GUI application and provide the entry fi as a parameter (value) to the connection.
4. Connect the aboutToCloseWidget event of the external GUI application to the object of the entry field and provide the customerNameData attribute of the external GUI application as the parameter to the connection.
5. Save the GUI application as member **GUISHR3**.
6. Test the application. Change the data in the entry field and then click on the p Change the data in the application that is opened. Close the window and see the the initial window change.

1.7.7.2.2 Pass by Reference

Pass by reference in a procedural language requires you to pass a pointer to the function that you call. Using this pointer to the memory location of the data, the function can update the real data. There is no requirement for the function to return the data to the calling application because it was working on the same data anyway. The danger is of course that the calling application has less control over what the function is allowed to do. If different functions are allowed to process asynchronously, you also have the danger of both functions changing the same data simultaneously.

Pass by reference requires a mechanism to access and store the pointer of a part. Each part in VisualAge Generator has an attribute that contains its memory location: self.

VisualAge Generator also offers a part that enables you to store a pointer: the variable part. The variable part is one of the parts in the Models category of the parts palette. It only has one attribute: self. This attribute contains the memory location of the part to which the variable part is connected. Thus the variable part should always be connected to another part. If it is not, it does not point anywhere and might cause incorrect references when you try to use it. Because a variable part contains the memory location of the part to which it is connected, a variable part can point to any VisualAge Generator part, not just a working storage record.

Once you have connected the variable part to another part by connecting the self attribute of that part to the self attribute of the variable part, you can perform any actions on the variable part that you would be able to do on the part itself, for example, access its attributes (data). Since it points to the part it uses the same memory location upon which to perform the action. The variable part has become a way of looking at and acting upon the part.

But how can you make a connection to an action that is not defined for the variable part. As mentioned before, one of the only attributes of a variable part is the self attribute. The variable part can choose an entry from the Connect window that contains unlisted action/attribute/event.... Figure 39 shows the window that is shown with an entry field in which you can type the name of the action, attribute, or event you want to perform, change, or use. As long as you are sure that the feature is implemented in the part to which the variable part is connected, this will work. If it is not, your connection may fire but it will not trigger an action or change an attribute.

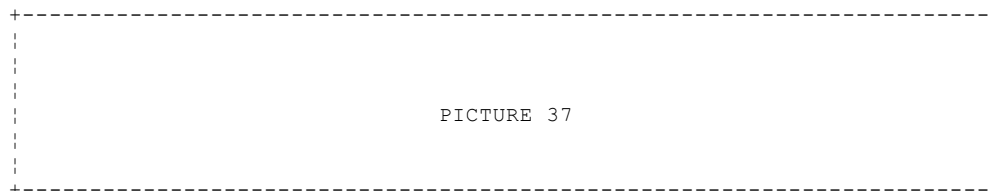


Figure 39. Window for Making a Connection to a Variable Part

VisualAge Generator also provides you with an option that enables you to make a variable part look just like a VisualAge Generator member. Choose **Build features based on member...** from the Object menu of the variable part. This selection brings up a window with an entry field. Type the name of the VisualAge Generator member whose interface you want the variable part to use. Once you have set this feature and you look at the interface of the variable part, it will look just like the interface of the member on basis of which you have build its features. Using this option does not mean that the variable part actually points at this member. If you connect the variable part to the self attribute of a push button, it points to the push button, and you can click it.

HINT	
	You may have noticed that the variable part looks a lot like a torn-off attribute. E
	represented graphically as parts surrounded by brackets. A torn-off attribute is inc
	variable part. It points to one of the attributes of the part from which it has been
	You can also easily create connections by dropping the same type of part the variable
	represents on the free-form surface. Tear off the self attribute. Make your connect
	this tear-off and, when you are finished, delete the original part.

GUI PROGRAMMING EXAMPLE: PASS BY REFERENCE

This example builds on the GUI programming example Promoting a Feature in topic 1.7.7. To pass data using the pass by reference technique, do the following:

1. Open GUI application **GUISHR1**.
2. Add a variable part to the free-form surface of GUI application. Promote the self of the variable part as customerName.
3. Choose **Connect...** from the Object menu of the variable part. Select the entry un attribute... from the list of attributes. Type in "object" as the attribute name. It is in lower case. Make the connection to the object attribute of the entry field. A warning message box will be shown. Accept the warning and continue to make the connection.
4. Save the GUI application as **GUISHR4**.
5. Open GUI application **GUISHR3**.
6. Select **Change member name...** from the Object menu of the external GUI application free-form surface and select **GUISHR4** from the drop-down list. A warning message box will be shown. Accept the suggestion and change the subpart name.
7. Change the connections from the external GUI application (**GUISHR4**) to use the customerNameData attribute instead of the customerName attribute.
8. Change the connections from the entry field on the window part in which the external application is embedded to using self instead of object.
9. Save the GUI application as **GUISHR5**.
10. Test the application. Change the data in the entry field and then click on the promote button. Change the data in the application that is opened. Notice that the data now also appears in the original entry field. Close the window.

1.7.7.3 Global Variables

Data can also be shared by using VisualAge Generator tables. A VisualAge Generator table can be used as a container of global variables. Because the VisualAge Generator table is resident in memory and can be shared by all GUI applications, connections are not necessary.

Remember that events indicating that the data has changed are not signaled. You have to request the newest data every time you use it. See "VisualAge Generator Table Characteristics" in topic 1.7.5.1 for more information and examples.

1.7.8 Data Format in a GUI Application

Several attributes of a part describe the kind of data it represents (such as the data type of a data item) and how data is formatted inside the part. The most important is the data type of the part. Table 11 shows the relationship between the data type of visual parts and the data types of data items. The entries indicate which combinations will not cause runtime errors independent of what input values, permitted by the visual part, that the user provides.

Table 11. Data Type Relationship between GUI Application Parts and VisualAge Generator Data Items.

D Default data type automatically assigned when Quick Form function is used.
O Optional data type that can be set.

Cells left empty represent an invalid relationship.

GUI Application Part Data Type	VisualAge Generator Data Item Data Type					
	Char	DBCS	Mixed	Hex	Nonnumeric (no decimals)	(c
<none>	O			O	O	
Boolean	O					
Date	O		O			
DBCS	O	D	O			
Integer	O				D	
Monetary	O				O	
Number	O				O	
String	D		D	D		
Time	O		O			

To customize the data type, click on the **Customize...** push button on the settings window of the part. The data type can be customized to include:

- ☐ A definition of how the data, once entered, is represented to the end user
- ☐ A range of valid values
- ☐ A default value

Figure 40 shows an example of the customization window for a part with data type Integer.

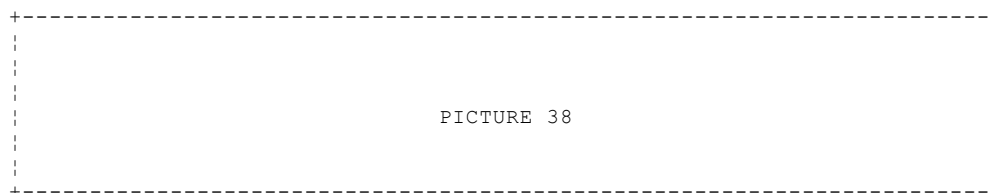


Figure 40. The Input Field Specifications of an Entry Field

If the data, as provided by the user or the program, does not conform to the specifications, either data type or range, the part will show "*** error ***" in the field.

1.7.9 Summary

The use of data is an important part of any application. In VisualAge Generator data is stored in data items. A working storage record is a collection of data items.

Data items either have a numeric or nonnumeric data type. Data items are only relevant as part of a structure such as a working storage record or a VisualAge Generator table. The definition of the data item can be either global (shared by all uses of the global data item) or local (defined within the context of the structure).

Data structures can be used to group data items. This grouping provides for both data management and GUI application functions such as Quick form. Data structures can also be torn off in a GUI application so that direct access to the lower level data items and the occurring data items is possible. An ordered collection can be torn off of a working storage record or a &VGt..

Multiple techniques are available for sharing data between GUI applications. Each technique works differently and has subtle operational attributes that will affect your choice for sharing data.

1.7.10 What You Should Now Be Able to Do

You should now be able to:

- ☐ Understand the pros and cons of using certain data types
- ☐ Relate the data types to some basic data types in programming
- ☐ Understand basic data type operations in VisualAge Generator procedural language like typecasting, finding etc.
- ☐ Understand how data is stored in a VisualAge Generator in working storage record
- ☐ Define a prolog for a record
- ☐ Differentiate between length and bytes
- ☐ Understand the relationship between the description of a data item and the label used in the Quick Form
- ☐ Understand the defaults used by the Quick Form action
- ☐ Know that you should use the DATAITEM data attribute to connect to an entry field.
- ☐ Understand the difference between the string and object of an entry field
- ☐ Understand what aggregation is and its client/server limitations
- ☐ Understand how aggregated data can be represented
- ☐ Understand the purpose of structuring data (typecasting and substringing)
- ☐ Understand level-77 data items
- ☐ Use structured items in your application
- ☐ Understand what to tear off
- ☐ Understand the limits of a working storage record (in size and dimensions)
- ☐ Understand a solution for multiple dimensional arrays
- ☐ Understand the attributeName attribute of a part
- ☐ Understand that the best way to fill a list is to use getFieldsStartingAt:to:
- ☐ Understand how to use parameters to share data between parts
- ☐ Use a working storage record to share data within one GUI application and between GUI applications
- ☐ Understand the public interface of GUI applications and how it can be used to share data between different GUI applications.

1.8 Chapter 8. Procedural Logic in VisualAge Generator

Although you build GUI applications mainly by using graphical techniques, it is also possible to define and use procedural code in a GUI application. In fact, there is no reason not to use procedural code in a GUI application, except perhaps for a graphical method purist. Procedural logic in a GUI application offers additional function, reduces the complexity of the application, and improves performance at runtime.

Subtopics

- 1.8.1 Procedural Parts in VisualAge Generator
- 1.8.2 Accessing Data from Procedural Logic
- 1.8.3 Performing Actions Using Procedural Logic
- 1.8.4 Summary
- 1.8.5 What You Should Now Be Able to Do

1.8.1 Procedural Parts in VisualAge Generator

Three different parts contain VisualAge Generator procedural logic:

- ☐ Callable Function
- ☐ Process Member Part
- ☐ Statement Group Member Part

The callable functions represent server (or "called batch") applications. Server applications can run on any platform, including the local workstation and remote workstations and hosts, and be written in any language. The server applications that are called have to be defined in a linkage table so that VisualAge Generator can identify them and call them with the correct convention and middleware option. Because errors can occur in the call to server applications, it is best to call them from a statement group or process, so that you can check for any error messages that might appear after the call (see Chapter 12, "GUI Error Handling" in topic 2.3 for more details on error checking in a GUI application).

A statement group and a process, when used in a GUI application, are identical. A process in a GUI application cannot have a process option other than execute because no I/O is allowed in the GUI application. Therefore the only difference is the color of the icon and the fact that the statement group can be used in the TEST statement that checks whether a value occurs in a VisualAge Generator table. Although there are no other differences between a statement group and a process with an execute process option, it is recommended that you choose one member type that will always be used in your GUI applications. This will facilitate building precise member lists in VisualAge Generator.

The process and statement groups are nonvisual parts that are placed on the free-form surface of a GUI application. To execute the process or statement group, use the execute action--one of the features available for procedural nonvisual parts. The logic in processes and statement groups used in GUI applications is generated into Smalltalk for runtime implementation just like the other parts and visual connections defined as part of a GUI application.

1.8.2 Accessing Data from Procedural Logic

In procedural logic within a GUI application you cannot perform I/O to a database or file system; that can only be done by server applications.

In processes and statement groups in GUI applications you can only access and change data in working storage records or VisualAge Generator tables. After the process has been executed, the data is changed in the GUI application and causes the events related to the attribute values to be signaled.

The working storage records and VisualAge Generator tables that you access have to be present on the same free-form surface as the process or statement group that uses them. A variable part used to represent a working storage record or a VisualAge Generator table in a GUI application does not support procedural access to the working storage record or VisualAge Generator table member.

If you do require some logic to access data in a record that is passed to your GUI in a variable, you can either:

- ☐ Signal an event from your GUI to the GUI that contains the real copy of the record and ask that GUI to perform the logic for you

or
- ☐ Pass the statement group part defined in the GUI that owns the data to your GUI as a variable as well, then execute the statement group indirectly, just like you access the data.

HINT

You can use the **Build Features Based on Member** function from a statement group part a from a record member.

You can access the data from a working storage record or a VisualAge Generator table by referencing the data item. You reference the data item by placing the name of the data item in the procedural code. The data item can be qualified as being part of a specific working storage record by prefixing the data item with the name of the working storage record and separating the two with a period. This is the preferred method of referencing a data item because it can occur in multiple working storage records on the free-form surface of the GUI application. You do not have to refer to the data item differently if it is part of a substructure.

If you want to refer to a certain element in an occurring structure, you have to suffix the data item name with the index of the element you want to access enclosed in parentheses. The index can itself be a reference to a data item although you cannot qualify this reference.

Although you can place an SQL row record on the free-form surface, we recommend that you not do so. First of all, if the SQL row record is passed to an application as a parameter, the application cannot use the same SQL row record as a process option for an I/O operation. In addition, the use of an SQL row record makes your representation dependent on the database type used by your application.

1.8.3 Performing Actions Using Procedural Logic

Procedural code in a VisualAge Generator GUI application enables you to use all of the features of the VisualAge Generator procedural language that do not require you to use process options to perform I/O on a file or database system. Using procedural code you can cause actions to be taken on parts in your application or algorithms to be implemented.

Subtopics

1.8.3.1 Perform Request

1.8.3.2 Algorithms

1.8.3.1 Perform Request

A process or statement group cannot directly perform actions on other parts in the GUI application, except for working storage records and VisualAge Generator tables. However, the perform request feature of VisualAge Generator GUI applications enables you to indirectly perform actions on other parts.

Perform request is the term used for a programming construct that enables you to build up a GUI action that has to be taken in procedural code. It literally means: "perform a request for an action." The request is passed by storing it in a working storage record structure. By providing this structure at the moment of execution, the request as described in the working storage record is executed.

A request consists of the part to be influenced (identified by its name), the attribute of the part to be changed or the action to be performed on the part, and any parameters that should be passed to the connection.

This information is represented in a working storage record. The working storage record can be any one as long as it contains the following structure:

```
10 REQUEST-STRUCTURE
12 PARTNAME
12 ACTION
12 PARAMETER
```

The exact levels as stated are not required as long as the structure contains three data items. You are also free to choose the names of the data items. The data items must be defined as character but can be of any length as long as they can contain the required information. In fact you are building up a connection and defining its characteristics.

You can pass multiple parameters in the structure by substructuring the PARAMETER data item. You can also choose to make the structure contain multiple requests by setting the number of occurrences for the structure as a whole to more than 1.

Because a working storage record is used, you can set the values using a process. Remember that the VisualAge Generator GUI builder is case sensitive, so you should insert data with the correct case to the structure. Values keep their case in procedural logic if they are enclosed by double quotes.

The action is executed by choosing an event (often the has executed of the process that filled the working storage record) and connecting it to the performRequest action of the GUI application part that can be accessed from the free-form surface. This creates a dashed line that expects requestObject as its parameter. Connect the REQUEST-STRUCTURE attribute of the working storage record to the connection, and your action is executed when the event is signaled.

While the perform request approach is powerful and flexible, quite a bit of overhead is involved in the implementation of the performRequest action. See "Conditional Logic" in topic 1.9.2 for a review of all your options and "Perform Request Considerations" in topic 2.4.3.6 for a study of the performance implications for the perform request technique.

| GUI PROGRAMMING EXAMPLE: CHANGING THE COLOR OF A WINDOW, USING PROCEDURAL LOGIC

```
|
|
| To open a message window when an error has occurred in a piece of procedural code, do
| following:
|
| 1. Open a new GUI application
|
| 2. Add a push button to the window.
|
| 3. Add a working storage record (PERFREQ) and a process (COLORWINDOW) to the free-form
|    Define the working storage record with the following structure:
|
|          10 PERFREQ-STRUCTURE      char 180
|          20 PARTNAME                char  60
|          20 ACTION                  char  60
|          20 PARAMETER               char  60
|
| 4. Code the following process:
|
|          PERFREQ.PARTNAME = "Window";
```

```
PERFREQ.ACTION = "backgroundColor";  
PERFREQ.PARAMETER = "Blue";
```

Window is the name of the part to be affected, backgroundColor is the action you perform on the part (in this case changing the value of the attribute), and Blue value that needs to be passed to the connection.

- 5. When the push button is clicked, the process should be executed. Connect the clicked of the push button to the execute action of the process.
- 6. Connect the has executed event of the process to the performRequest action of the surface. Pass it the PERFREQ-STRUCTURE attribute of the working storage record as parameter.
- 7. Test the application (save the GUI application as **PRCTST1**).

1.8.3.2 Algorithms

Algorithms can be built with processes or statement groups using all the language features that are available in VisualAge Generator (for example, conditional statements, loops, and tests). When defining a process, you can use a template to find the statement you want to use. Choose **Edit** and then **Insert after with template...** when in the process or statement group editor. This brings up the window shown in Figure 41.

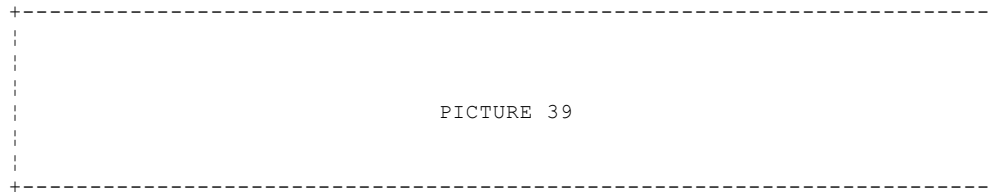


Figure 41. Defining a Procedural Statement by Using a Statement Template Window

A process or statement group can perform or execute another process or statement group just like in a VisualAge Generator server application. The process or statement group that is called does not have to be present on the free-form surface. If it is present, it bears no relationship to the actual process or statement group that is executed. For instance, the has executed event will not be signaled from a process on the free-form surface named ABC when the ABC process is performed by another process.

1.8.4 Summary

The callable function part enables you to add server applications to your GUI application. A process with execute as its process option and a statement group can be used to write procedural logic that is part of the GUI application.

Procedural logic in a GUI application cannot perform I/O. It can only access record member parts that are on the free-form surface of the same GUI application. Procedural logic can only directly change working storage records and VisualAge Generator tables in the GUI application. You can use the perform request to indirectly execute other actions in the GUI application.

Algorithms can be built up from the statements provided by the VisualAge Generator procedural language. By using the template to define statements, you can interactively define syntactically correct statements.

1.8.5 What You Should Now Be Able to Do

You should now be able to:

- ☐ Understand the purpose of using procedural logic in VisualAge Generator GUI applications
- ☐ Build a process as part of a GUI application
- ☐ Understand the limitations of procedural logic in a GUI application, including accessing data and performing actions through a perform request.

1.9 Chapter 9. Visually Building Logic

Algorithms in applications are built up from sequences, iterations, and conditional statements. In this chapter we cover ways of implementing these basic concepts by using the visual programming techniques provided by VisualAge Generator. We discuss the implementation of certain actions that are a combination of other actions and the possibilities for conditionally executing actions based on the state of the system. We describe iterating actions a number of times or until a certain condition is satisfied.

Figure 42 shows the general structure of an event-driven application using sequences, iterations, and conditional statements. Processing begins with a user event, which starts an action that consists of sequences, iterations, and conditionals. Processing ends when the application has reached a reenterable state.

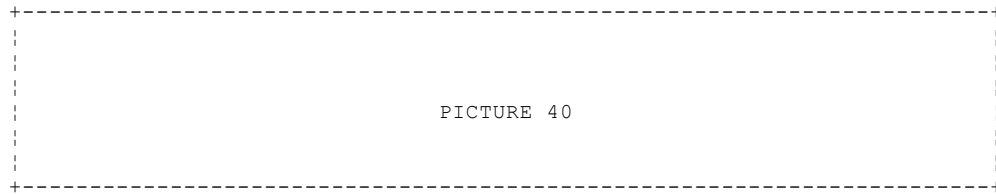


Figure 42. Structure of an Event-Driven Application

Subtopics

1.9.1 Sequencing Actions

In a procedural programming language, you can tell the system what to do next, thus creating a certain flow. This flow allows easy programming for algorithms but makes it difficult to program an application that is controlled by the user. In an event-driven environment, there is no such thing as the next action to be taken except if it results from the same event. This section provides you with a set of rules on the sequence in which actions are executed by VisualAge Generator.

Subtopics

1.9.1.1 Order of Events

1.9.1.2 Creating Actions

1.9.1.1 Order of Events

Most user actions cause a series of events to occur. These events are often signaled in a certain order. In this section we discuss the main user actions, show which combinations of events are signaled, and analyze the order in which events occur based on the action performed by the user. After you have read this section, you will be able to select the appropriate VisualAge Generator event to trigger actions when building applications.

Subtopics

1.9.1.1.1 Basic User Actions

1.9.1.1.2 The Order in Which a GUI Is Built and Destroyed

1.9.1.1.1 Basic User Actions

The events that occur when a user takes a certain action are listed below. The most common user actions are explained and a method for analyzing other actions is given. Although all events that occur are presented, not every event has to be connected to an action.

Clicking a Push Button: The following event occurs when you click a push button either as a user action or using the click action:

```
01 clicked (7)
```

Enabling a Part: The following event occurs when you enable a part with the enable action:

```
01 enabled
```

Opening a Window: The following events occur chronologically when you open a window by using the openWidget action:

```
01 aboutToOpenWidget
02 aboutToMapWidget
03 resized
04 openedWidget
05 mappedWidget
06 gettingFocus
```

Closing a Window: The following events occur chronologically when you close a window:

```
01 closeWidgetRequest
02 aboutToCloseWidget
03 closedWidget
04 destroyedPart
```

The closeWidgetRequest event occurs only when the user uses the system menu to close the window.

The aboutToCloseWidget and closedWidget events do not get signaled if the window is closed by using the destroyPart action.

The destroyedPart event shows up only if the destroyPart action is used to destroy the window.

Editing an Entry Field: The following events occur chronologically when you enter data into an entry field:

```
01 gettingFocus
02 object
03 string
04 userInputConvertError
05 losingFocus
```

The object and string events occur only when the value in the entry field has changed. The value changes at each keystroke when **Notify change on each keystroke** has been set in the settings of the entry field. This value changes when the field loses focus only if this indicator has not been set.

The userInputConvertError event is signaled instead of the object and string events when a data type has been set for the entry field other than "(none)" and the data type is violated.

Executing a Process: The following events occur chronologically when a process ends its execution:

```
01 data items in working storage records that were changed
02 has executed
```

The order in which the changes in the data items occur as events is not predefined. The order cannot be guaranteed.

Changing Data Items: Consider a working storage record with the following generic structure:

```
RECORD
  Level   Name
```



```
10    LEVELONE
20    LEVELTWO
30    LEVELTHREE
30    THREEEXTRA
20    TWOEXTRA
10    ONEEXTRA
```

The level 10 item can have multiple occurrences. This does not change the behavior as described here.

To access all of the data, you would need to tear off parts of the working storage record. In this case it has been torn off as three separate parts, which are referred to as:

1. The working storage record
2. Tear-off of LEVELONE
3. Tear-off of LEVELTWO from the tear-off of LEVELONE

In order to base actions on changes in the data, it is important to know which events occur if a data item is changed. The occurrences of events obey the following rules:

- ☐ The change of a data item causes both the data and the self event of the data item to be signaled.
- ☐ Changing a data item, such as LEVELTWO, that contains other data items causes all of the substructured data items to be changed and thus their corresponding events to be signaled.
- ☐ If the data item is part of a substructure, the data event is signaled for all of the data items of which it is part.
- ☐ Changing a data item causes the data event of the record to be signaled.

Figure 43 shows the hierarchy of data items and the changes that are caused if one of the data items is changed.

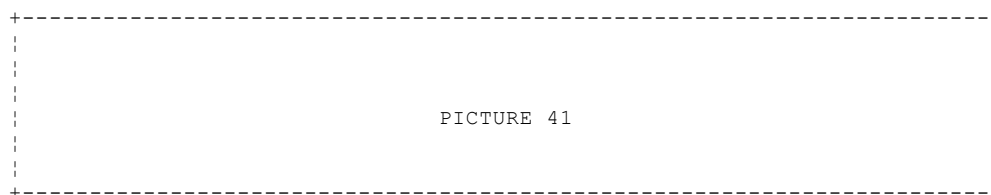


Figure 43. Hierarchy of Data Items and the Effects of a Data Item Change

There are two exceptions to the rules:

1. Level-77 data items

Because level-77 data items hold their own data and do not really belong to the record, changing their data does not have an effect on the record's data and vice versa.

Modifying a level-77 data item in a logic member does not cause the record data to signal.

Modifying the whole record (using, for example, the SET record EMPTY statement) causes the signaling of all record data items but not the level 77 data items attached to the record.

2. Callable functions (server applications)

Because the context in which callable functions execute is outside the domain of the GUI application, the GUI application cannot know which data items might have been modified by logic in the callable functions. Therefore, if a record is passed as a parameter, the whole record gets signaled (that is, all non-level-77 data items get signaled).

The chronological order of the changes is unpredictable. This is true for changes that occur in a process and changes that occur visually. The latter tend to follow the order as set in the **Reorder connections from** window.

+-----+
| HINT |
+-----+

	You can create your own event traces by creating an application where you connect all relevant events to a dummy action. Test the application with the trace set on and an trace to see the sequence in which the events occur. By using the name of the dummy filter in the trace, you only get the relevant trace entries.

- (7) The clicked action of a push button is usually preceded by a set of events for the part that had the focus before the click. If a text field had the focus, you will get the string, object, and loosingFocus events signaled from the field before the clicked action is signaled for the push button.

1.9.1.1.2 The Order in Which a GUI Is Built and Destroyed

A primary part (for example, a window) usually has a number of subparts. These subparts are created before the primary part in which they are held is created. Any events based on this creation occur before the `aboutToOpenWidget` event of the primary part.

The order in which separate parts are created depends on the order in which they were put onto the primary part during definition and not on the ordering or layout of these parts on the primary part. The parts defined first are also created first. The order corresponds to the tabbing order of the parts in the window and can be viewed by using the parts list. By changing the tabbing order the order of creation is changed.

If a part is created, the aboutToOpenWidget event for the part occurs when the part in which it is contained is opened (but before the aboutToOpenWidget event of the part in which it is contained fires). Every visual part that has the aboutToOpenWidget event as "widget" in VisualAge Generator can be thought of as a generic term for a visual part. Parts that are put on the free-form surface do not fire the aboutToOpenWidget event.

Before the parts on the primary part have been created and opened, all attribute-to-attribute connections within the GUI application trigger, but they only trigger if the attribute values of the parts concerned are not synchronized.

GUI PROGRAMMING EXAMPLE: UNDERSTANDING THE ORDER OF CONNECTION TRIGGERING

Build the following GUI to understand the ordering of aboutToOpenWidget events:

1. Open a new GUI application.
2. Put a toggle button and a push button on the window.
3. Connect the aboutToOpenWidget event of the window to the set action of the toggle button. Connect the aboutToOpenWidget event of the push button to the clear action of the toggle button.
4. Test the application (save the GUI application as **VSLTST1**) and see whether the toggle button gets set or not.
5. Turn on the ITF tracing, open a Trace log window, and restart the GUI application. The trace log should show you the order in which connections were fired.

A parts needs to be created before other parts can communicate with it. Creating a part is a different action on a part than opening it. The difference is explained in more detail in Chapter 11, "Building VisualAge Generator Parts" in topic 2.2 No attribute values of these parts can be set from outside the part until the part has been created.

The order in which the primary part is closed is not opposite to the order in which it is created. The part itself, the GUI application, is closed first followed by the primary part and all the parts contained in the primary part. These parts are closed in the order in which they were put onto the primary part and not in reverse order of which they were created.

1.9.1.2 *Creating Actions*

In this section we examine the behavior of events and actions in VisualAge Generator.

Subtopics

- 1.9.1.2.1 Event Handlers
- 1.9.1.2.2 Event Trees
- 1.9.1.2.3 Sequencing Events
- 1.9.1.2.4 Passing Parameters
- 1.9.1.2.5 Triggering Events from Other Connections

1.9.1.2.1 Event Handlers

Events in VisualAge Generator have handlers. A handler is something that is connected to an event. An action is a type of handler. Handlers execute atomically and some of them generate additional events (for example, click -> clicked, enable -> enabled, execute -> hasExecuted). This explains the default event combinations as described in "Basic User Actions" in topic 1.9.1.1.1.

Some events are generated synchronously, and some, asynchronously. In case of a synchronous event, the event signals its handlers before the next handler from the parent event is dispatched. If the event is signaled asynchronously its handlers are put on a queue and dispatched at the earliest opportunity. It is very difficult to predict "the earliest opportunity" because it depends on many factors. Asynchronous events are dispatched in this way for usability reasons. For example, the clicked event is deferred so that the data from text fields can be committed before the push button handlers are processed.

Figure 44 shows the basic structure of an event and its handler and the difference between asynchronous and synchronous handling of the event.

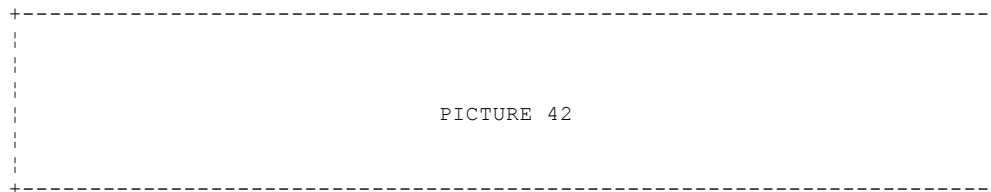


Figure 44. Event and Its Handler

In fact, when you write an application, you are writing handlers for certain events. As stated above, the handlers should be atomic, and you should consider whether the events they dispatch should be synchronous or asynchronous.

1.9.1.2.2 Event Trees

Now that we know about event handlers and their synchronous or asynchronous posting of events, let us look at the sequence of events.

An event can cause an action to be performed. The action can cause its own set of events to be performed, and so on. One complete sequence of actions based on one event is called an *event tree*. Like most trees, the event tree can have branches when along the way an event again causes multiple actions to be taken. There are three basic event sequence characteristics:

- The order in which events occur from the initiating part depend on the order of the connections as set in the **Reorder connections from** window. If the event has been promoted and triggers actions in the GUI application in which it is embedded, these actions occur after the events within the current GUI application are signaled.
- Actions can be seen as traversing a tree (see Figure 45). An action traverses the tree until it encounters the end of a branch, as long as the events are dispatched synchronously. It then starts traversing the next branch at the last split it encountered. This is true for most types of actions.

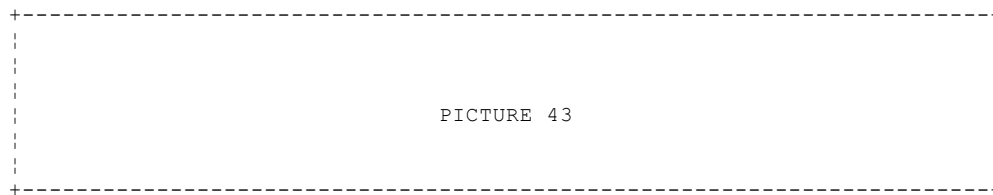


Figure 45. Traversal of the Event Tree

- If asynchronous actions (such as click of a push button) are used, this sequencing scheme is broken, and the system goes to the next action in the queue, placing the action that should be performed after the click in the queue.

1.9.1.2.3 Sequencing Events

In an event-driven system, it is not evident which event will occur next because all events are essentially asynchronous. To force a certain event to occur next, you can trigger an action that will cause an event to be signaled that is dispatched synchronously. This event can then be used for the next action.

+-----+ <u>HINT</u> +-----+	
+-----+	+-----+
	If the part that causes the event to be signaled is used only for this purpose, it is
	use a label and connect to its <u>enable</u> action and use its <u>enabled</u> event. This connect
	consumes the least system resources.
+-----+	+-----+
+-----+	+-----+

Sequencing events like this is one of the ways to ensure that they happen in the expected order. If there is a relationship between two paths, you have to make sure that the dependency is correct. Figure 46 illustrates this point.

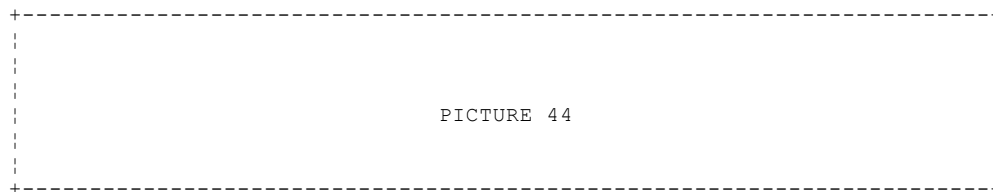


Figure 46. Sequences of Events

1.9.1.2.4 Passing Parameters

The concept of synchronously and asynchronously dispatched events also has an effect when providing parameters for a connection. You have to ensure that the connection containing the value of the parameter is fired before the connection requiring the parameter is fired. The GUI programming example that follows illustrates this point.

GUI PROGRAMMING EXAMPLE: USING ENTRY FIELD OBJECTS AS EVENTS	
	<p>To use entry fields as objects, do the following:</p> <ol style="list-style-type: none"> 1. Open a new GUI application. 2. Place two entry fields on the window. Uncheck the Signal events on each keystroke button in the Settings window of the entry fields. This ensures that a change to contents of the part occurs only when the field loses focus and the value has changed. 3. Connect the <u>object</u> event (be sure to choose the event and not the attribute) of the entry fields to the <u>backgroundColor</u> attribute of the window. 4. Provide the <u>object</u> attribute as the parameter to the connection. 5. Test the application (save as VSLTST2). <p>Notice that when you change the text in the entry field to "Blue," for instance, of the field, the color of the window does not change. Try that again. Change to "Red" and tab out. The color changes this time. The window turns blue. This behavior is caused by the fact that the change of the attribute value is performed after the event that the value has been changed has been signalled.</p> <p>The trace of this application is shown in Figure 47.</p> <p>The trace clearly shows that the passing of the parameter occurs after the event is signalled. This is even what you see when you choose Reorder connections from of the entry field. If you are not allowed to change the order in this situation.</p> <ol style="list-style-type: none"> 6. Edit the GUI application, remove the connection providing the parameter, and retest the application. Note that the application now works as intended. But why? <p>You may have noticed that when the first connection was made (entry field <u>object</u> to window <u>backgroundColor</u> attribute) it did not have a dashed line. It had a solid line. This may be telling us that the triggering event automatically includes the associated parameter. If so, this is one less connection that is required.</p>

00001 GUI Event: Text1 (#object) --> Window (#backgroundColor)
00002 GUI Event: Text1 (#object) <--> (Text1,object --> Window,backgroundColor) (#value)
00003 GUI Event: Text1 (#object) --> Window (#backgroundColor)
00004 GUI Event: Text1 (#object) <--> (Text1,object --> Window,backgroundColor) (#value)

Figure 47. Trace of Connection Triggering for GUI Programming Example

When you use the change of an attribute as an event in VisualAge Generator, the new value of the attribute is automatically passed as the first parameter on the connection. This is a very powerful technique in that it reduces the number of connections required and causes fewer events to occur in the system. This characteristic of connections is valid for all attributes that are used as events except when the events stem from changes to a data item in a working storage record or VisualAge Generator table.

1.9.1.2.5 Triggering Events from Other Connections

One very good way of forcing an order to a set of connections is to trigger the start of one connection on the basis of when another connection has finished.

The result attribute of a connection will trigger once the connection has completed. By using the result attribute as an action trigger you can start other connections in sequence.

| GUI PROGRAMMING EXAMPLE: ORDERING CONNECTIONS USING A RESULT ATTRIBUTE

To order connections using a result attribute, do the following:

1. Create a new GUI member.
 2. Add a push button to the window.
 3. Add a second window on the GUI free-form surface.
 4. Place an entry field in the second window.
 5. Add a working storage record (**RCDTST1**, created in the Chapter 7, "Working with Data" example) and a process (**VSLPROC**) to the free-form surface.
 6. Add this code to the **VSLPROC** process: **MOVE 'PROC DATA' TO RCDTST1.ITEM1;**
 7. Connect clicked of the push button to openWidget of the second window.
 8. Connect clicked of the push button to execute of the process.
 9. Test the application (save as **VSLTST3**) and click on the push button.
- What happens? (You may have to adjust the relative execution speed of the ITF test record to change the order of processing. Change the speed to 7 and test again.)
10. Move the push button end of the connection between the push button and the window so that the connection starts with the result of the connection between the push button and the process.
 11. Test this application and click on the push button.

What happens?

Now the window should be opened **after** the process has executed. This removes the scroll bar associated with the arrival of the contents of the entry field after the window has opened.

1.9.2 Conditional Logic

Programming involves making choices. Choices can be made by including conditional code. In this section we describe how VisualAge Generator supports conditional coding in building GUIs. We discuss the most well-known mechanism, the perform request, first. We then focus on using attributes as toggles and building conditional visual connections.

Subtopics

1.9.2.1 Perform Request

1.9.2.2 Data Item

1.9.2.3 Visual Conditional Logic

1.9.2.1 Perform Request

The perform request mechanism can be used to conditionally execute a connection (see "Perform Request" in topic 1.8.3.1 for an explanation of the perform request mechanism). Because the procedural code supports the IF statement, you can change the values with which you fill the perform request structure according to the result of the condition.

GUI PROGRAMMING EXAMPLE: USING PROCEDURAL CODE AND A PERFORM REQUEST ACTION TO OPEN A MESSAGE WINDOW

To use perform request logic to open a message window when an error has been detected of procedural code, do the following:

1. Create a GUI application with an entry field and a push button on the window. Make the push button close the window.
2. Promote the object attribute of the entry field as this is where you want the message displayed. Save the GUI application as **MSGWIN**.
3. Create another GUI application. Put the **MSGWIN** GUI application, a working storage record (**PERFREQ2**), and a process (**PR-ERRORCHECK**) on the free-form surface. Add an entry field and a push button to the window.
4. Define the **PERFREQ2** working storage record so it has the following structure:

10	PERFREQ-STRUCTURE	char	180
20	PARTNAME	char	60
20	ACTION	char	60
20	PARAMETER	char	60
10	MESSAGE	char	60
5. Code the following for process **PR-ERRORCHECK**:


```
IF MESSAGE NOT BLANKS;
  PERFREQ2.PARTNAME = "MSGWIN";
  PERFREQ2.ACTION = "openWidget";
ELSE;
  PERFREQ2.PARTNAME = " ";
  PERFREQ2.ACTION = " ";
END;
```
6. When the push button is clicked, the process should be executed.
7. The contents (object) of the entry field should be connected to the **PERFREQ2** attribute MESSAGE data.
8. The **PERFREQ2** attribute MESSAGE data should be connected to the promoted feature (object of the entry field) of the MSGWIN GUI application.
9. Connect the has executed event of the process to the performRequest action of the surface. Pass it the PERFREQ-STRUCTURE attribute of the working storage record as the request object of the connection to the performRequest action.
10. Test the application (save as **VSLTST4**). Click on the push button (nothing happens). Enter some data in the entry field in the first window and click on the push button (what happens?).

1.9.2.2 Data Item

A connection can be triggered by using a data item in a working storage and changing the value (conditionally) in a process. The change in the value signals an event from the change in the attribute value of the data item after the process has executed.

Because you are using an attribute, you must be careful that no other changes to the data item or the working storage record of which it is part cause the event to be fired. Therefore you should use level-77 data items for this purpose. Changes to the working storage record of which they are part do not influence the data item.

+-----+
| GUI PROGRAMMING EXAMPLE: USING A DATA ITEM TOGGLE TO OPEN A MESSAGE WINDOW
+-----+

+-----+
| This example shows the same error window as in the previous example, now using a data
| toggle for opening the window.
|
| 1. Save the previous example with a new name (**VSLTST5**).
|
| 2. Create a new working storage record (**TRIGGER**) with the following data items:
|
| 10 MESSAGE char 60
| 77 MSGTRIGGER char 1
|
| 3. Replace the perform request working storage record with the new working storage.
| easiest way to do this is to hold down the Altkey and click on the working storage
| with mouse button 1. Now change the name from **PERFREQ** to **TRIGGER**.
|
| 4. Create a new process (**TR-ERRORCHECK**). Give it the following code:
|
| IF TRIGGER.MESSAGE NOT BLANKS;
| MSGTRIGGER = "Y";
| END;
|
| 5. Replace the previous process with the new process. The easiest way to do this is
| down the Altkey and click on the process with mouse button 1. Now change the name
| **PR-ERRORCHECK** to **TR-ERRORCHECK**.
|
| 6. Delete the perform request connections (process to free-form surface and record to
| connection).
|
| 7. Create a connection between the **TRIGGER** attribute MSGTRIGGER data and the openWin
| of the **MSGWIN** GUI application.
|
| 8. Test the application (save first).
+-----+

1.9.2.3 Visual Conditional Logic

Instead of having to create conditional logic every time, it would be nice if there was a part that you could ask whether a certain condition is true or not. The part should allow you to set the condition, clear the condition, and check whether the flag is set, returning true or false depending on the current status of the flag.

GUI PROGRAMMING EXAMPLE: CREATING A CONDITIONAL VISUAL LOGIC FLAG PART

To create and use a visual part that allows you to implement conditional logic without the need to implement the perform request or data triggering mechanism on every occurrence, do the following:

1. Create a new GUI application. Replace the window with a label part that has a labelString value of "Flag." This will be the primary part just to indicate what part we are using when using the part.
2. Add five more label parts with these labelString values:
 - ☐ setFlag
 - ☐ clearFlag
 - ☐ checkFlag
 - ☐ true
 - ☐ false
3. Add a variable part and name it "Condition."
4. Make the following connections:
 - a. Connect the enabled events of the setFlag and clearFlag labels to the self attribute of the "Condition" variable part.
 - b. Connect the enabled event of the checkFlag label to the "unlisted action ..." attribute of the "Condition" variable part.
 - c. Connect the self attribute of the true label to the value attribute of the condition between the setFlag label and the "Condition" variable part.
 - d. Connect the self attribute of the false label to the value attribute of the condition between the clearFlag label and the "Condition" variable part.
5. Promote the following features using a promoted part feature name that matches the labelString for each part:
 - a. The enable action of the setFlag, clearFlag, and checkFlag labels
 - b. The enabled attribute of the true and the false labels
6. Save this GUI application as member FLAG.

.

.

.

.

.

.

The part is now ready to be used. You can set or clear the flag. You can ask the part whether the flag is set. If it is, the true event is signalled; if it is not, the false event is signalled.

1. Create a new GUI application. Add the created FLAG GUI application to the free-form window of the new GUI application.
2. Put two push buttons on the window: "Set" and "Clear." Connect the clicked event of the push buttons to the setFlag and clearFlag actions of the FLAG GUI application, respectively.
3. Put a third push button on the window: "Check." Connect the clicked event of this button to the checkFlag action of the embedded GUI application.
4. Connect the true event of the embedded GUI application to the backgroundColor action of the window. Give it "Green" as its hard-coded parameter.
5. Connect the false event of the embedded GUI application to the backgroundColor action of the window. Give it "Red" as its hard-coded parameter.
6. Save this GUI application as member FLAGTST.

	7. Test the application. Click on Set and then Check, Clear and then Check, what ha

There are many places in a GUI application where a conditional part, accessible and integrated with the visual logic, can make the programming task easier to accomplish. You may want to change the `labelString` and/or the part name of the embedded part so that your use of the part is obvious.

1.9.3 Iterations

Iterations are useful for repeating an action a number of times until a certain criterion is satisfied. In this section we explain a number of methods to achieve this using visual connections. Since the end of a loop depends on a condition, there is some relation to the paragraph about conditional logic.

Subtopics

1.9.3.1 Iterator

1.9.3.2 Other Iterations

1.9.3.1 Iterator

Loops are often used to go through all of the elements in an array and perform some kind of action on them. Within VisualAge Generator the ordered collection is a kind of array (see "Ordered Collections" in topic 1.7.6 for a description of ordered collections). The ordered collection contains a subpart, called the *iterator*. The iterator is a pointer to one of the elements in the array. It has methods that enable you to reset it to point to the first element in the array and actions that enable you to point to the next element in the array. The element to which it points at any moment can also be accessed by the iterator.

To access the iterator choose **Tear-off attribute** from the Object menu of the ordered collection. Choose iterator from the list of attributes and drop the part on the free-form surface.

The iterator has the following interface:

Actions:

<u>iterate</u>	Advances the cursor to the next element in the collection, if there is a next element, and answers a Boolean indicating whether the cursor was advanced
<u>next</u>	Advances the cursor to the next element in the collection, if there is a next element, and gives a pointer to the next element as a result
<u>nextLine</u>	Advances position past next line delimiter and returns all elements up to the delimiter. This action works if you are iterating over a string or a collection that contains two consecutive elements that correspond to the character CR (carriage return) followed by the character LF (linefeed).
<u>peek</u>	Returns the value of the next item in the array but does not move the pointer
<u>reset</u>	Resets the pointer to the beginning of the array

Attributes:

<u>atEnd</u>	Boolean indicating whether the end of the array has been reached
<u>contents</u>	Contains the collection that is being iterated over
<u>current</u>	Provides a pointer to the item currently pointed to
<u>isEmpty</u>	Boolean indicating whether the array is empty
<u>size</u>	Integer indicating the number of elements stored in the array.

Events:

The events are similar to the changes of the attributes.

<u>atEnd</u>	Change of the Boolean indicating whether the end of the array has been reached
<u>contents</u>	Contains the collection that is being iterated over
<u>current</u>	Change of the pointer to the item currently pointed to
<u>isEmpty</u>	Change of the Boolean indicating whether the array is empty
<u>size</u>	Change of the integer indicating the number of elements stored in the array

```
+-----+
| GUI PROGRAMMING EXAMPLE:  USING THE ITERATOR |
+-----+
```

```
+-----+
|                                     |
|                                     | To use the iterator to close the windows that were opened by the system, do the follow
|                                     |
|                                     | 1. Create a new GUI application. Put an entry field on the window. Save the window
|                                     |
```


VSLTST6.

2. Create another GUI application. Add the **VSLTST6** GUI application as an external GUI application on the free-form surface three times.
3. Add two push buttons to the window: **Open** and **Close**.
4. Add an ordered collection to the free-form surface.
5. Connect the **clicked** event of the **Open** push button to the openWidget action of each of the three external GUI applications. Connect the clicked event of the **Open** push button three times to the add: action of the ordered collection. Provide the self attribute of each of the different GUI application as a parameter on each of the connections. (Connect self to anObject.)
6. Tear off the iterator attribute of the ordered collection. Tear off the current attribute of the ordered collection from the torn-off iterator attribute.
7. Connect the clicked event of the **Close** push button to the reset action of the iterator. This resets the iterator to the beginning.
8. Drop a label on the free-form surface. Connect the clicked event of the **Close** push button to the enable action of the label. Connect the enabled event of the label to the next action of the iterator. This puts the cursor on the next element in the array.
9. Connect the enabled event of the label to the closeWidget action of the torn-off iterator attribute of the iterator (you will need to choose unlisted action... and type in closeWidget yourself).
10. Connect the closedWidget event of the torn-off current attribute of the iterator (using the unlisted event... option from the Connect window) to the enable action of the label. This starts the next iteration.
11. Test the application (save as **VSLTST7**).
12. Open all the windows by clicking on the **Open** push button and close them all by clicking on the **Close** push button.

1.9.3.2 *Other Iterations*

As an iteration is ultimately controlled by a conditional event, the methods for building conditional logic ("Conditional Logic" in topic 1.9.2) can also be used to build iterating actions. Start an action and as a last part of this action trigger a piece of conditional logic that determines whether the action should be performed again.

1.9.4 Summary

Algorithms usually consist of sequences, iterations, and conditional statements. These basic concepts can be built in VisualAge Generator using different techniques.

Events have handlers, which can be actions. These handlers may perform a number of actions and in turn cause a number of events to be signaled. These events can occur asynchronously, which means they are put last on the stack of events to be handled, or synchronously, which means they are put first on the stack.

When writing your own handlers, it is important to make sure that they are atomic. They should have a begin and an end with a predefined state. In addition you should clearly define the events that are signaled by the handler and whether they are synchronous or asynchronous.

Conditional execution of actions can be achieved by using a perform request, a data item as a toggle, or visual conditional logic. A perform request uses a data structure that contains the GUI action to be performed. This data structure can be filled by using a process. The use of a data item as a toggle is based on the principle that the data item's event occurs only when it is changed in the process. Conditional visual logic is based on actions that do not always cause events to be signaled.

An ordered collection is one of the main candidates on which to perform iterations. It has a subpart called the iterator which can be used to loop through the elements of the array. The loop can be controlled by checking whether the end of the array has been reached.

The techniques described to build conditional code can also be used to build iterations. Iterations are basically the re-execution of an action depending on whether a certain condition has been met.

1.9.5 What You Should Now Be Able to Do

You should now be able to:

- ☐ Build a VisualAge Generator GUI application consisting of atomic, reentrant event handlers
- ☐ Understand and be able to recognize the difference between asynchronous and synchronous events
- ☐ Describe the order of events as they occur in the system based on the definition
- ☐ Understand the specific characteristics of changes in data items
- ☐ Use changes in attributes as events with automatically provided parameter
- ☐ Know the order in which events occur when a window is opened or destroyed
- ☐ Understand and be able to apply the different types of conditional logic in your application
- ☐ Understand and be able to use the different types of iterations in your application

2.0 Part 2. Run

This part of the book covers the issues that would come up when you build a GUI application using the techniques described in Part 1. After you read this part you will be able to develop VisualAge Generator GUI applications effectively.

Subtopics

- 2.1 Chapter 10. Advanced GUI Features
- 2.2 Chapter 11. Building VisualAge Generator Parts
- 2.3 Chapter 12. GUI Error Handling
- 2.4 Chapter 13. Performance

2.1 Chapter 10. Advanced GUI Features

In this chapter we discuss some of the advanced features provided with VisualAge Generator GUI builder.

Subtopics

- 2.1.1 Container Details
- 2.1.2 Drag and Drop
- 2.1.3 File Accessor Part
- 2.1.4 Closing a Window
- 2.1.5 Summary
- 2.1.6 What You Should Now Be Able To Do

2.1.1 Container Details

The container details is a list that can contain a number of columns--the contents of which can be edited if desired. The container details displays the column data in a format that is easy for the user to manage, especially when the information displayed contains more than one data type. The container details also provides a solution for seamless scrolling: users do not have to manually ask for more data to be retrieved from the database for the current list they are viewing.

Subtopics

- 2.1.1.1 Creating a Container Details
- 2.1.1.2 Features of a Container Details
- 2.1.1.3 Scrolling Data
- 2.1.1.4 Editing in a Container Details

2.1.1.1 Creating a Container Details

A container details is created when you create a Quick Form of a substructured item that has multiple occurs; for nonsubstructure occurring items, Quick Form produces a list. You can also create one yourself by selecting the container details from the Lists category and adding the required columns manually. When you use a Quick Form, the settings of the column are adjusted automatically for the data item that will be shown.

The default way to fill a container details with data would be to have the connection from a working storage record structure to the items attribute of the container details. This default connection does not perform well, however, because it has to figure out how many data items have actually been filled by comparing each to the default value. A better method of filling a container details is by providing an array with a known size (start and end) and providing this information to the container details.

A substructured array in a working storage record, when torn off, has a getFieldsStartingAt:to: action. Use an event, such as when you have finished fetching the data, to trigger this action. This creates a connection that requires two parameters: firstIndex and lastIndex. The firstIndex parameter can be set to 1 (assumes that the array always starts in the first occurs). The lastIndex parameter can be set to the number of rows (the last occurs instance that will be included) that you want to provide to the container details. By connecting the result attribute of this connection to the items attribute of the container details, the container details is filled with the selected data when the event that causes the getFieldsStartingAt:to: action to be triggered is signaled.

The container details fills efficiently when you use the above technique. A relationship is also created between the working storage record and the container details. The action passes the container details a pointer to the data in the working storage record. So any changes you make in the container details are also represented in the working storage record.

2.1.1.2 Features of a Container Details

A list is used to present data to the user as well as to enable the user to select one or more entries from the list for further processing. In this context a number of possible selection techniques are supported by the container details, such as single select, extended selection, and multiple select. Because you can potentially select multiple entries, the container details does not have a selectionIndex attribute. Instead it has a selectionIndices attribute, which, if torn off, gives you an ordered collection. The first attribute of the ordered collection can be thought of as the selectionIndex.

The container details provides the user with information spread over a number of columns. Often the columns spread out past the right-hand boundary of the container details, thus requiring the user to scroll right. The data that is shown in the container details often contains one or more data items that uniquely define the row. The user will want to keep seeing this data even though he or she scrolls right. The lockedColumns feature, which can be set in the settings or at runtime, defines the number of columns that should always remain visible, starting at the left-most column.

From the settings you can modify the look and feel of the container details. You can set the rows to have an etched look, making them gray with a backdrop. You can also make the container details read only if the user is not allowed to make changes in the data or when you are using the container details only as a selection mechanism.

2.1.1.3 Scrolling Data

The container details provides a mechanism for seamless scrolling called packetRequested. The mechanism enables you to fill a container details with a certain number of elements. When you scroll down the list and reach the point where you run out of data, as provided in the first "packet," the event packetRequested is signaled. This event can be used to get additional data (call a server) to fill the next packet of information.

To enable the function of the packetRequested for the container details, you have to set the packetEnabled attribute of the container details to true. This will ensure that the packetRequested event gets signaled when a new packet is required. Before doing this, however, you have to set the total number of rows you are going to fetch in the totalRows attribute of the container details. This ensures that the scroll bar of the container details is set up correctly, to look like the container details already contains the total number of rows of data, although in fact it does not.

When both the packetEnabled and totalRows attributes have been set, you are ready to start using the packet request capabilities of the container details. Use the packetRequested event as your signal to go and fetch the next set of data. The packetSize attribute can be used to predefine the size of the packets so that they match the size of the working storage record array you use to hold the data. Instead of adding this data to the container details directly, however, you have to use the packet attribute.

By tearing off the packet attribute, you get access to its features, which include:

startRow The start position of the page of data at which the scrollbar is currently located

endRow The end position of the page of data at which the scrollbar is currently located

dataRows The rows of data to be shown in the container details. (The number of rows should be equal to the difference between the endRow and startRow attribute.)

The difference between the endRow attribute and the startRow attribute determines the size of the packet of data. This is dependent on the size of the container details on the window.

Now, when the packetRequested event is signaled, you can get the packet of data that corresponds to the startRow until endRow of your result set. The result of this fetch should be placed in the dataRows attribute of the tear-off of the packetRequested event. The technique you use here should be the same technique used for entering data into the container details in the most efficient way; in other words, use the getFieldsStartingAt:to: action.

This mechanism works fine if you can predict the values in your result set between startRow and endRow. If you are using a relational database system, this can be quite difficult. You could, however, write a query that does exactly that using a WHERE clause similar to that which will be used to get the data:

```
SELECT
:key
FROM table T1
WHERE where-clause
AND :startrow =
  (SELECT COUNT(
primary key)
FROM table T2
WHERE where-clause)
ORDER BY order-clause
```

On basis of the resulting value of *key*, you can build up the rest of your result set.

You will have to weigh the performance of this query against the requirement for seamless scrolling.

```
+-----+
| GUI PROGRAMMING EXAMPLE: SEAMLESS SCROLLING
```

```
+-----+
|                                     | This example uses the features of a container details to create an application that a
```

seamless scrolling through a set of data.

1. Create a new GUI application. Create a working storage record, **PACKETWS**, with the structure and put it on the free-form surface:

10	TOTALROWS	bin	9	
10	STARTROW	bin	9	
10	ENDROW	bin	9	
10	PACKETSIZE	bin	9	
10	PACKETNUMBER	num	3	
10	ROWS	char	21	occurs 50
15	NUMBER	bin	9	
15	TEXT	char	17	
20	TEXT-P1	char	14	
20	TEXT-P2	num	3	
77	INDEX	bin	9	

2. Make a Quick Form from the ROWS attribute. This results in a container details attribute-to-attribute connection that is created.
3. Add an entry field and a push button to the window. Connect the object attribute entry field to the TOTALROWS data attribute of the working storage record.
4. Connect the clicked event of the push button to the:
 - a. packetEnabled attribute of the container details. Clear the toggle button parameter. You may have to set and then clear the toggle button to get the connection link solid.
 - b. totalRows attribute of the container details and provide the TOTALROWS data attribute of the working storage record as a parameter.
 - c. packetEnabled attribute of the container details. Set the toggle button parameter.

.

.

.

.

.

.

5. Create a statement group named CDV-GET-DATA with the following statements:

```

/* Note: Uncomment the marked (/*C) lines and the packet size
/* will vary with the visible size of the CDV.
/* Endrow and startrow values automatically vary
/* by visible size of the CDV.

/* Determine packet number (for understanding)
PACKETNUMBER = PACKETNUMBER + 1;

/* Determine the size of the packets
; /*C PACKETSIZE = 1 + ENDROW - STARTROW;

/* Check if not greater than the number of occurs
/* in the working storage
; /*C IF PACKETSIZE GT 50;
PACKETSIZE = 50;
; /*C ENDROW = STARTROW + 50 - 1;
; /*C END;

/* Fill the array with data
INDEX = 0;
WHILE INDEX LT PACKETSIZE;
  INDEX = INDEX + 1;
  NUMBER(INDEX) = STARTROW + INDEX - 1;
  TEXT-P1(INDEX) = "Packet Number";
  TEXT-P2(INDEX) = PACKETNUMBER;
END;
;

```

6. Tear off the ROWS attribute of the working storage record.
7. Tear off the packet attribute of the container details.
8. Connect the packetRequested event of the container details to the execute action statement group.
9. Connect the has executed event of the statement group to the getFieldsStartingAt attribute of the torn-off attribute (ROWS) of the working storage record. Using the setting of this connection give the firstIndex attribute parameter the value 1. Connect the

data attribute from the working storage record to the lastIndex attribute of the result attribute of the connection to the dataRows attribute of the packet attribute.

10. Connect the STARTROW data and ENDROW data attributes of the working storage record to the startRow and endRow attributes of the torn-off packet attribute.

.
.
.

.
.
.

11. Test the application (save the GUI application as **CDVPAKET**).
12. Enter a number to represent the total rows in the container details in the entry click on the push button. Scroll down in the container details. You will see the CDV-GET-DATA statement group being triggered to get the next packet.

It helps if you set ITF watch points on these data items from the PACKETWS working storage record:

TOTALROWS	STARTROW	ENDROW	PACKETSIZE	INDEX
-----------	----------	--------	------------	-------

Try a new total rows value and click on the push button again.

Try changing the size of the container details. (If you connect the container details window using the container details layout settings you can do this during testing). dynamic change in packetSize if you have removed the comments from the statements with **/*C**).

2.1.1.4 Editing in a Container Details

Often the container details is used to enable the user to change the data in multiple rows of data. The editable character of a container details, or a column in a container details, can be changed at runtime by setting or clearing the value for the editable attribute. There is an important consideration, however. When you use a container details in situations where you do not know how much data it will hold or this amount cannot be fetched from the server at once, there are special considerations if you allow editing of the container details contents. You need to consider that when you are changing the data you will need to cache the changes and implement a solution to store the changes once the user indicates they want to save the data.

If you edit the data in a column of the container details, the data in the associated working storage record is updated automatically. The aboutToBeginEdit and aboutToEndEdit container details column events can be used as triggers for checking the values of the data. You could also be ready for a change to any cell in the container details by using the event cellValueChanged. Remember that, as with entry fields, the data gets updated only when the cell loses focus. The cell does not lose focus when the user uses a menu to start the save action, so you will have to make sure that the cell loses focus yourself (for example, by explicitly putting the focus on another part). The container details and the columns in it have the same features as an entry field for validating information (see "Data Validation Errors" in topic 2.3.2 for a more detailed explanation of validation). See the *VisualAge Generator GUI User's Guide and Reference* for further explanation of the features of a container details.

2.1.2 Drag and Drop

Drag and drop is a very powerful feature in VisualAge Generator that can also simplify the design of your user interfaces. An object is dragged from a source and dropped on a target. The drop is valid only if the target part accepts the type of object that is dropped on it.

Whether or not a part supports drag and drop is indicated in the settings of the part. You can change the characteristics only during definition of your GUI application, not at runtime. For both drag and drop there are three options: move, copy, or link:

- Move--When both the source and target support move, you can move data from the source to the target (default operation when you allow both move and copy).
- Copy--When both the source and target support copy, you can move data from the source to the target. (Press the Ctrl key to copy if both move and copy are supported.)
- Link--creates a link between the source and the target, making the parts point to each other. (Use Ctrl+Shift and mouse button 2 to make a link.)

GUI PROGRAMMING EXAMPLE: IMPLEMENTING DRAG AND DROP

This example shows you how to drag and drop between two list boxes.

- ☐ Create a new GUI application.
- ☐ Put two list boxes on the window beside each other. Change the settings of the left one to allow you to drag (both move and copy) and the right one to allow you to drop (both move and copy).
- ☐ Add some hard coded values to the left list box.
- ☐ Test the application (save the GUI application as **DRAGDROP**).

Try both moving and copying some data (select an item first). Notice that the icon indicates whether dropping is allowed and shows you whether you are about to copy the item. When you are copying the selected item, you see a plus sign in the icon being dragged.

Each part that allows drag and drop has a feature called `dragDropSpec`. You can tear off this attribute. This gives a number of additional features that allow you to react to certain events in the drag and drop process. See the *VisualAge Generator GUI User's Guide and Reference* for further explanation of these events.

2.1.3 File Accessor Part

The file accessor part enables you to access files on the client workstation without server applications. The file accessor part can be found in the Models category. It has a very basic structure. You can select a file that brings up the standard operating system file dialog box. The file you select is stored in an attribute. Once you have access to the file, you can read and write from it, using the buffer attribute as an intermediate storage area.

GUI PROGRAMMING EXAMPLE: USING THE FILE ACCESSOR PART	
	In this example we build a simple editor, using the file accessor part.
	<input type="checkbox"/> Create a new GUI application. Put a multi-line edit part on the window.
	<input type="checkbox"/> Put two push buttons on the window. Label one Open... , one Save .
	<input type="checkbox"/> Put a file accessor part on the free-form surface. Connect the <u>buffer</u> attribute of the file accessor part to the <u>object</u> attribute of the multi-line edit part.
	<input type="checkbox"/> Connect the <u>clicked</u> event of the Open.. push button to the <u>selectFile</u> action as well as the <u>read</u> action of the file accessor part.
	<input type="checkbox"/> Connect the <u>clicked</u> event of the Save push button to the <u>write</u> action of the file accessor part.
	<input type="checkbox"/> Test the application (save the GUI application as FILEACC).

2.1.4 Closing a Window

Users can use a number of methods to close a window. You have some control over most of the methods; you can ask users whether they are sure they want to exit or ask them whether they want to save the data. If the user uses a push button or menu item to close a window, you still have an option, after posing these questions, to send or not send the `closeWidget` message to the window. You cannot stop the user from double-clicking on the system menu to close the window. This is a function provided by the operating system.

Double-clicking on the system menu will cause the `closeWidgetRequest` event to go off, however. This event can be used to show a message window to the user. Since the closing of the window is in process, though, this action has to be canceled. The window has a `cancelCloseRequest:` action for this purpose. This cancel request needs information about the initial close request. This can be passed to it by passing the `closeWidgetRequest:` event as the parameter (`confirmStruct`) to the connection.

You can also directly connect the `closeWidgetRequest` to the `cancelCloseRequest` action. This causes the window never to close when the user double-clicks on the system menu. Using the `closeWidgetRequest` event, you can now implement your own closing procedure, after which you issue a normal `closeWidget` action.

GUI PROGRAMMING EXAMPLE: USING A MESSAGE BOX AS A CONFIRMATION DIALOG	
	This example implements a message box with the simple question as to whether the user wants to close the window.
	<ul style="list-style-type: none">□ Create a new GUI application. Put a message prompter on the free-form surface. message prompter to provide a question. Enter "Are you sure you want to close the window?" as the message text. Make sure you select the Yes, No radio button.□ Connect the <code>closeWidgetRequest</code> event of the window to the <code>cancelCloseRequest:</code> action of the window. This stops the close and allows you to decide what to do.□ Connect the <code>closeWidgetRequest</code> event of the window to the <code>prompt</code> action of the message prompter.□ Connect the <code>okYesRetry</code> event of the message prompter to the <code>closeWidget</code> action of the window.□ Test the application (save the GUI application as CNFRMSG).
	Test both the negative and positive answers to the question stated in the message box.

2.1.5 Summary

VisualAge Generator provides a number of facilities that can enhance the GUI.

The container details provides a tabular view of data. It enables the user to scroll through the data, change the size of the columns, and edit directly within the table. As a developer you can control the look and feel of the part, the scrolling process, and any editing that is done by the user. In using the container details, be aware of the implications that large sets of data, and allowing the user to change them, can have on your system design.

Drag and drop enable you to simplify your GUI by allowing the user to pick up data from one location and drop it on another without providing for some form of flow between the windows. The drag and drop process is controllable through the settings of the parts to which it applies.

The file accessor part enables you to access data from an operating system file through a buffer. The user is provided with the standard operating system file dialog to select a file. Once selected, read, and changed, you can write the data back to the file.

A user can always close a window by double-clicking on the system menu. The closeWidgetRequest event allows you to intervene by, for example, showing a message window. Using the cancelCloseRequest:, you can then decide whether or not to close the window.

2.1.6 What You Should Now Be Able To Do

You should now be able to:

- ☐ Use the container details and understand both its features and limitations.
- ☐ Use drag and drop as a facility to enhance your user interface.
- ☐ Provide support for accessing operating system files in your application, using the file accessor part.
- ☐ Control the closing of windows.

2.2 Chapter 11. Building VisualAge Generator Parts

Embedded GUI applications can be used to reuse parts in an application and reduce its complexity. In this chapter we look at creating embeddable parts and making them communicate with each other. We also look at dynamic programming, that is, creating parts at runtime.

Subtopics

2.2.1 Embeddable Parts

2.2.2 Dynamic Programming

2.2.3 Summary

2.2.4 What You Should Now Be Able To Do

2.2.1 Embeddable Parts

VisualAge Generator allows you to reuse parts that are recurrently used in your application development effort. For example, if you always want to show address information in a similar way regardless of whether you are talking about an employee or a customer, you could create a part that contains all fields relevant to the address information on a form and store it as a separate GUI application. When you want to add or show address information on a window, you just include a GUI application, and you have the required function.

Subtopics

- 2.2.1.1 Creating Embeddable Parts
- 2.2.1.2 Adding Parts to the Palette
- 2.2.1.3 Part-to-Part Communication
- 2.2.1.4 Sharing Data

2.2.1.1 Creating Embeddable Parts

Embeddable parts are VisualAge Generator GUI applications that do not have a window but some other visual part as their primary part. The embeddable parts cannot be run by themselves but should always be part of another GUI application. In the GUI application that embeds the embeddable part, only the primary part of the embeddable part is visible. All parts within the embeddable part that must be visible to the end user should therefore be in the primary part of the embeddable part.

Figure 48 shows the structure of an application that includes embeddable parts.

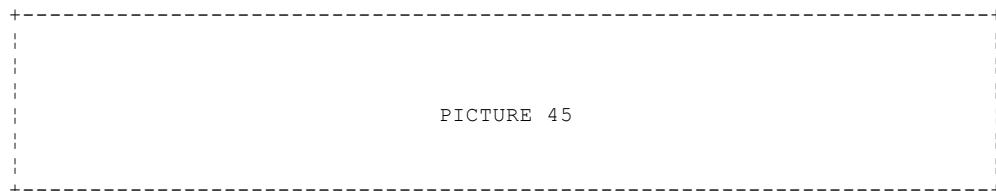


Figure 48. Application Structure With Embeddable Parts. The structure shown is just an example. There are no rules about how you mix embeddable parts and visual parts. You could also include embeddable parts directly on the free-form surface if the part is not used as part of the visual user interface.

You create an embeddable part just as you would create any other GUI application. Create a GUI member. Because the primary part will not be a window, the first thing you can do is delete the window. If this embeddable part will be used to construct a visible part of the user interface, determine which part the end user is going to see and place it on the free-form surface. If it is the first part to be placed on the free-form surface, it will automatically become the primary part. If not, you can explicitly make it the primary part by choosing **Become primary part** from the Object menu.

The embeddable part is in many ways similar to an external GUI application that has a window as its primary part. Like the external GUI application, the embeddable part provides a view of certain data and can react to user actions. For example, a user could click on a push button that is implemented in the embeddable part. You can use this event to clear all fields in the embedded part.

An embeddable part can have other embeddable parts implemented in it. The amount of nesting is limited only by the amount of memory and performance of the system (see Chapter 13, "Performance" in topic 2.4). A GUI application can even be nested in itself.

To embed an embeddable part in a GUI application, choose **Options Add GUI application...** from the menu of the GUI builder of the part in which you want to embed the part. Choose the member of your embeddable part from the list and click on **OK**. Move the crosshairs to the location where you want the part to appear and click mouse button 1. You could also drop the part by using the GUI application member part from the part palette. In this case the GUI builder assumes the part to be an external GUI application. You have to drop it on the free-form surface, before you can choose the GUI application from a list.

GUI PROGRAMMING EXAMPLE: <u>DEFINING AN EMBEDDABLE PART</u>	
In this example we create an embeddable part that shows address information. The part has four fields for the street, city, state and zipcode.	
1. Create a new GUI member and delete the window.	
2. Create a working storage record, ADDRESS-WS , that contains the following data items:	
10 ADDRESS	char 92
20 STREET	char 50
20 CITY	char 30
20 STATE	char 2
20 ZIPCODE	char 10
3. Put a form on the free-form surface. Make sure it is the primary part by looking at the Object menu. If the menu option Become primary part is not available, the form is not the primary part.	

4. Put the **ADDRESS-WS** working storage record on the free-form surface. Tear off the data item. Quick Form the self attribute of this tear-off onto the form. Align to make the part aesthetically viewable.

.
.
.

.
.
.

5. Add a push button with the label "Clear" to the free-form surface (the user will click this push button). Connect the clicked event of the push button to the **ADDRESS** clear attribute of the working storage record. This connection will clear the contents of the structure in the working storage record and therefore clear the entry fields. Since we do not provide a parameter to the connection, blanks are automatically passed.

6. Add a process to the free-form surface called **SET-ADDRESS** and provide it with the following code:

```
ADDRESS-WS.STREET = "My Street 1";
ADDRESS-WS.CITY = "My Home Town";
ADDRESS-WS.STATE = "MS";
ADDRESS-WS.ZIPCODE = "1111111111";
```

7. Save the part as **ADDRESS** and close the GUI application definition.
8. Create a new GUI member. Choose **Options Add GUI application...** from the menu and select **ADDRESS** from the list of GUI members.

9. Place the embeddable part on the window.

10. Test the application (save the GUI application as **ADDRWIN**).

You will see no difference between the application using an embeddable part and an application that would have placed the entry fields directly on the window. You have made your embeddable part reusable and hidden some of the complexity of showing address information inside the embeddable part.

2.2.1.2 Adding Parts to the Palette

The embeddable part can be added to the palette just like any other VisualAge Generator part. This can be especially useful if the embeddable part is going to be used extensively within your development team. Choose **Options Modify palette** from the menu and choose either create a new category or add the part to an existing category.

To create a new category, choose **Add new category**. Give the category a meaningful name and choose bitmaps to represent the category when it is closed and opened. If you want to delete a category, select it and choose **Options Modify palette Delete category** from the menu. You can only delete categories that are not part of the default product configuration.

Choose **Add new part** and select the GUI member to be added to the palette from the list. You can also set an icon to be used as the icon in the parts palette. Choose the category to which the part should be added and click on **Promote**. If you want to delete a part, select it and choose **Options Modify palette Delete part** from the menu. You can only delete parts that are not part of the default product configuration.

Once you have added a part to a category, you can access it like any other part in VisualAge Generator GUI builder. The GUI member must be in your concatenation of MSLs in order to use it from the palette.

Information on the palette is stored in a file called EZE2PRT.CAT. You can find the file in the VisualAge Generator work directory, usually \EZERDEV2, and you copy it to the other workstations to ensure that they get the same palette. When you install a new version of VisualAge Generator, the file will be overwritten. Make a backup of the file before installation. When new parts are added to a new release of VisualAge Generator, they will be accessible only if you use the new EZE2PRT.CAT as provided with the product. You will have to manually add your own parts to the palette again.

In a larger development environment the solution suggested in "Part Settings" in topic 1.2.3.4 has more validity. Create a separate GUI application with all of the parts that should be available to everyone in your development team with the correct settings. Allow users to copy and paste to and from the clipboard to get the required parts with the correct settings.

2.2.1.3 Part-to-Part Communication

Like any other part in VisualAge Generator the embeddable parts you create have an interface. If you do not do anything special, the interface will be identical to the interface of the primary part of the embeddable part. If you want to be able to perform other actions on parts of the embeddable part from within the GUI application in which it is embedded, you have to define the actions as being part of the interface of the embeddable part.

Determine the actions that you would like to be able to perform on the embedded object. Next, determine how this function can be provided. Determine the part and the corresponding action, event, or attribute within the embeddable part that allows you to implement this function. Choose **Promote part feature...** from the Object menu of this part. Figure 49 shows the window from which features are promoted to the interface of the embeddable part.

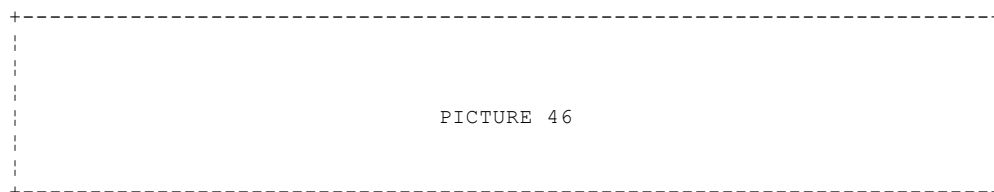


Figure 49. Window for Promoting Features to the Interface

Select the action, attribute, or event from the list and provide it with a meaningful name (see "Promoted Features" in topic B.2.3.2 for more information about naming promoted features). Click on the **Promote** push button. The feature is added to the list of promoted features. If the embeddable part is embedded in a GUI application, the embeddable part's Connect menu contains all of the promoted features.

To remove promoted features, select them from the list of promoted features of a part and click on the **Remove** push button. To rename promoted features, you must remove them and then promote them again.

Defining the interface of a part is critical to its usability. The interface of reusable parts should be stable. One of the goals of using embeddable parts is to reduce the complexity of applications. The complexity should be hidden inside the part and should not be part of its interface.

GUI PROGRAMMING EXAMPLE: ADDING FUNCTION TO AN EMBEDDABLE PART	
	In this example we add the capability to access and clear the address data from the p which the embeddable part is embedded.
	1. First we have to determine which features allow us to access and clear the data:
	<input type="checkbox"/> Accessing the data is possible through the <u>data</u> attribute of the working stor <input type="checkbox"/> Clearing the data can be performed by clicking on the <u>Clear</u> push button. <input type="checkbox"/> Filling the working storage record with data is done by executing the SET-ADD process.
	2. Open the ADDRESS GUI application (created in Defining an Embeddable Part in topic
	3. Select Promote part feature... from the Object menu of the working storage record
	4. Select the <u>data</u> attribute from the list of attributes and promote it with a promo name of <u>addressData</u> . Close the promoted features window.
	5. Select Promote part feature... from the Object menu of the push button.
	6. Select the <u>click</u> action from the list of actions and promote it as <u>clear</u> . Close promoted features window.
	7. Select Promote part feature... from the Object menu of the process.
	8. Select the <u>execute</u> action from the list of actions and promote it as <u>fillData</u> . C promoted features window.
	9. Save and close the GUI application definition.
	10. Open the ADDRWIN GUI application (created in Defining an Embeddable Part in topic
	11. Add two push buttons to the window underneath the embeddable part. Label the fir "Fill" and the second, "Clear." Change the names of the push buttons to "pbFill"

"pbClear," respectively.

12. Connect the clicked action of the Fill push button to the fillData attribute of the embeddable part.
13. Connect the clicked action of the Clear push button to the clear attribute of the embeddable part.
14. Save and test the application. Clicking on **Fill** should set the address data; clicking on **Clear** should clear it.

2.2.1.4 Sharing Data

Communication between parts involves the sharing of data. In this section we discuss a number of ways in which data can be shared among GUI applications.

Subtopics

2.2.1.4.1 Working Storage

2.2.1.4.2 Variable Parts

2.2.1.4.3 VisualAge Generator Tables

2.2.1.4.1 Working Storage

When two GUI applications contain a working storage record, data can be passed from one application to the other through an attribute-to-attribute connection. This makes data available in both GUI applications. If you make the connection unidirectional, one of the working storage records becomes read only.

GUI PROGRAMMING EXAMPLE: SHARING DATA BETWEEN GUI APPLICATIONS USING WORKING STORAGE RECORDS

This example moves data from one GUI application to another, using working storage records.

1. Open the **ADDRWIN** GUI application (created in Defining an Embeddable Part in topic 2.2.1.3 and modified in Adding Function to an Embeddable Part in topic 2.2.1.3).

2. Add the **ADDRESS-WS** to the free-form surface.

The **ADDRESS-WS** working storage record as used in both GUI applications are separate and can have different data.

3. Connect the data attribute of the working storage record to the addressData attribute of the embeddable part. This connection will keep the data of the two working storage records synchronized.

4. Save and test the application.

.

.

.

.

.

.

If you turn trace on, open a trace log window, and then restart the test run, clicking the **Fill** push button and then the **Clear** push button will cause the following trace:

```
00001 GUI Event: ADDRESS-WS (#ADDRESS) <--> ADDRESS of ADDRESS-WS (#self)
00002 GUI Event: ADDRESS of ADDRESS-WS (#CITY data) <--> Text2 (#object)
00003 GUI Event: ADDRESS of ADDRESS-WS (#ZIPCODE data) <--> Text4 (#object)
00004 GUI Event: ADDRESS of ADDRESS-WS (#STREET data) <--> Text1 (#object)
00005 GUI Event: ADDRESS of ADDRESS-WS (#STATE data) <--> Text3 (#object)
00006 GUI Event: ADDRESS-WS (#data) <--> ADDRESS (#addressData)
00007 GUI Event: ADDRESS-WS (#ADDRESS) <--> ADDRESS of ADDRESS-WS (#self)
00008 GUI Event: ADDRESS of ADDRESS-WS (#CITY data) <--> Text2 (#object)
00009 GUI Event: ADDRESS of ADDRESS-WS (#ZIPCODE data) <--> Text4 (#object)
00010 GUI Event: ADDRESS of ADDRESS-WS (#STREET data) <--> Text1 (#object)
00011 GUI Event: ADDRESS of ADDRESS-WS (#STATE data) <--> Text3 (#object)

00012 GUI Event: pbFill (#clicked) --> ADDRESS (#fillData)
00013 ++++++ ADDRESS 12-12-96 11:39:52 AM ++++++
00014 >>>> SET-ADDRESS
00015 ***** SET-ADDRESS *****
00016 >>>> 001 ADDRESS-WS.STREET = "My Street 1";
00017 ADDRESS-WS.STREET = "
00018 ADDRESS-WS.STREET = "My Street 1
00019 >>>> 002 ADDRESS-WS.CITY = "My Home Town";
00020 ADDRESS-WS.CITY = "
00021 ADDRESS-WS.CITY = "My Home Town
00022 >>>> 003 ADDRESS-WS.STATE = "MS";
00023 ADDRESS-WS.STATE = "
00024 ADDRESS-WS.STATE = "MS"
00025 >>>> 004 ADDRESS-WS.ZIPCODE = "111111111";
00026 ADDRESS-WS.ZIPCODE = "
00027 ADDRESS-WS.ZIPCODE = "111111111"
00028 GUI Event: ADDRESS of ADDRESS-WS (#STATE data) <--> Text3 (#object)
00029 GUI Event: ADDRESS of ADDRESS-WS (#ZIPCODE data) <--> Text4 (#object)
00030 GUI Event: ADDRESS of ADDRESS-WS (#STREET data) <--> Text1 (#object)
00031 GUI Event: ADDRESS of ADDRESS-WS (#CITY data) <--> Text2 (#object)
00032 GUI Event: ADDRESS-WS (#data) <--> ADDRESS (#addressData)

00033 GUI Event: pbClear (#clicked) --> ADDRESS (#clear)
00034 GUI Event: Push Button1 (#clicked) --> ADDRESS-WS (#ADDRESS data)
00035 GUI Event: ADDRESS-WS (#ADDRESS) <--> ADDRESS of ADDRESS-WS (#self)
00036 GUI Event: ADDRESS of ADDRESS-WS (#CITY data) <--> Text2 (#object)
```

```
|
|      00037 GUI Event: ADDRESS of ADDRESS-WS (#ZIPCODE data) <--> Text4 (#object)
|      00038 GUI Event: ADDRESS of ADDRESS-WS (#STREET data) <--> Text1 (#object)
|      00039 GUI Event: ADDRESS of ADDRESS-WS (#STATE data) <--> Text3 (#object)
|      00040 GUI Event: ADDRESS-WS (#data) <--> ADDRESS (#addressData)
|      00041 GUI Event: ADDRESS-WS (#data) <--> ADDRESS (#addressData)
|
|      The trace clearly shows that the two working storage records are synchronized four times:
|      initialization, when the data has been filled, and twice when the data has been cleared.
+-----+
+-----+
```

Passing data between working storage records through attribute-to-attribute connections creates overhead because the data in the working storage records must be synchronized. This movement of data is unnecessary because the data should always be the same. Chapter 13, "Performance" in topic 2.4 contains a more detailed description of the performance implications of different ways of sharing data.

part actually points at this member. If you connect the variable part to the self attribute of a push button, it points to the push button and you can click on it.

HINT

You can also easily create connections by dropping the same type of part the variable represents on the free-form surface. Tear off the self attribute. Make your connect this tear-off. When you are finished, delete the original part.

GUI PROGRAMMING EXAMPLE: SHARING DATA BETWEEN GUI APPLICATIONS, USING VARIABLE PARTS

This example moves data from one GUI application to another, using variable parts.

1. Save the **ADDRWIN** GUI application (last modified in Sharing Data between GUI Applications Using Working Storage Records in topic 2.2.1.4.1) with the new name **ADDRVAR**.
2. Delete the **ADDRESS-WS** working storage record from the free-form surface of the **ADDRVAR** application.
3. Add a variable part to the free-form surface of **ADDRVAR**. Choose **Build features by member...** from the Object menu of the variable part and enter ADDRESS-WS.
4. Edit the embedded GUI application **ADDRESS** and promote self of the **ADDRESS-WS** working storage record as address.

Remember when using a variable part you must provide it with the pointer to the pointer attribute (the self attribute) to which it should point. You could remove the addressData feature because you no longer need it in the **ADDRVAR** GUI application, but we will leave it because it is still referenced by the **ADDRWIN** GUI application.

5. Save the **ADDRESS** GUI application.
6. Connect self of the variable part you added to the newly created address feature of the **ADDRESS** embedded GUI application.
7. Save and test the **ADDRVAR** GUI application.

.

.

.

.

.

.

If you turn trace on, open a trace log window, and then restart the test run, clicking the **Fill** push button and then the **Clear** push button will cause the following trace:

```
00001 GUI Event: ADDRESS-WS (#ADDRESS) <--> ADDRESS of ADDRESS-WS (#self)
00002 GUI Event: ADDRESS of ADDRESS-WS (#CITY data) <--> Text2 (#object)
00003 GUI Event: ADDRESS of ADDRESS-WS (#ZIPCODE data) <--> Text4 (#object)
00004 GUI Event: ADDRESS of ADDRESS-WS (#STREET data) <--> Text1 (#object)
00005 GUI Event: ADDRESS of ADDRESS-WS (#STATE data) <--> Text3 (#object)
00006 GUI Event: Variable1 (#self) <--> ADDRESS (#address)

00007 GUI Event: pbFill (#clicked) --> ADDRESS (#fillData)
00008 ++++++ ADDRESS 12-12-96 12:02:18 PM ++++++
00009 >>>> SET-ADDRESS
00010 ***** SET-ADDRESS *****
00011 >>>> 001 ADDRESS-WS.STREET = "My Street 1";
00012 ADDRESS-WS.STREET = "
00013 ADDRESS-WS.STREET = "My Street 1
00014 >>>> 002 ADDRESS-WS.CITY = "My Home Town";
00015 ADDRESS-WS.CITY = "
00016 ADDRESS-WS.CITY = "My Home Town"
```

```

00017 >>>> 003 ADDRESS-WS.STATE = "MS";
00018 ADDRESS-WS.STATE = " "
00019 ADDRESS-WS.STATE = "MS"
00020 >>>> 004 ADDRESS-WS.ZIPCODE = "1111111111";
00021 ADDRESS-WS.ZIPCODE = " "
00022 ADDRESS-WS.ZIPCODE = "1111111111"
00023 GUI Event: ADDRESS of ADDRESS-WS (#STATE data) <--> Text3 (#object)
00024 GUI Event: ADDRESS of ADDRESS-WS (#ZIPCODE data) <--> Text4 (#object)
00025 GUI Event: ADDRESS of ADDRESS-WS (#STREET data) <--> Text1 (#object)
00026 GUI Event: ADDRESS of ADDRESS-WS (#CITY data) <--> Text2 (#object)

00027 GUI Event: pbClear (#clicked) --> ADDRESS (#clear)
00028 GUI Event: Push Button1 (#clicked) --> ADDRESS-WS (#ADDRESS data)
00029 GUI Event: ADDRESS-WS (#ADDRESS) <--> ADDRESS of ADDRESS-WS (#self)
00030 GUI Event: ADDRESS of ADDRESS-WS (#CITY data) <--> Text2 (#object)
00031 GUI Event: ADDRESS of ADDRESS-WS (#ZIPCODE data) <--> Text4 (#object)
00032 GUI Event: ADDRESS of ADDRESS-WS (#STREET data) <--> Text1 (#object)
00033 GUI Event: ADDRESS of ADDRESS-WS (#STATE data) <--> Text3 (#object)

```

The trace clearly shows that the memory address of the working storage record is passed once to the variable part. Thereafter the two parts point to the same memory location and further events are triggered to synchronize their content.

When a variable part is used to reference data, overhead is not created. A disadvantage of variable parts is that using the data in a working storage record to be accessed in a process or statement group requires the working storage record itself and not a variable part to be present on the free-form surface of the GUI application in which the process or statement group is implemented.

HINT

When you create a connection to a variable part that requires a parameter, you will not be able to provide a parameter to the connection. If you want to provide a parameter, first promote the attribute to the result attribute of the connection. Open the settings of the connection and in the entry field where it says result type in the name of the parameter that has been passed.

This will not work for attributes of working storage records, which have a different type. Any attributes you type in will be put inside single quotes. Attributes of working storage records are referenced by preceding them with a "#" sign; they are not surrounded by single quotes. In this case you have to use **Build features based on member...** to add a parameter to the connection or make the connection.

HINT

Often you will need embedded parts to communicate with the window in which they are contained. You can do this by passing the embedded part the self attribute of the parent (in this case the window). Promote the self attribute of the variable part in the embeddable part and then pass this to the self attribute of the window where it is embedded. In this way you can use the variable part as if it is the window in which the embeddable part is embedded.

2.2.1.4.3 VisualAge Generator Tables

VisualAge Generator tables can be used to share data between GUI applications. They represent a form of direct access to a common data store but do not support triggering actions based on a changed attribute event.

VisualAge Generator table data sharing support is discussed in "VisualAge Generator Table Characteristics" in topic 1.7.5.1.

2.2.2 Dynamic Programming

VisualAge Generator allows you not only to define applications during definition but also to add parts at runtime. This is called *dynamic programming*.

Creating parts at runtime is especially useful if you do not know which parts should be on the window or when you want to load parts into memory in the background to reduce the initial load time of the application. In this section we focus on ways to create parts at runtime.

Subtopics

2.2.2.1 Object Factory

2.2.2.2 Difference between Open and Create

2.2.2.3 Putting Parts on a Window

2.2.2.1 Object Factory

The Models category contains a part called the object factory, which is a factory for creating parts at runtime. Using an object factory to improve the performance of your system is discussed in more detail in Chapter 13, "Performance" in topic 2.4.

The settings of the object factory contain one main feature besides the part name: `instanceClassName`. The `instanceClassName` defines the type of part that should be created by the object factory. The field can contain either a VisualAge Generator GUI application name or the classname of a VisualAge Generator part. The `instanceClassName` can also be set dynamically by using a connection that changes its attribute value.

Once you have set the `instanceClassName` of the object factory, all attributes of the interface of the part that you are trying to create are automatically added to the interface of the object factory. By setting values for these attributes, a newly created part will have the values of these attributes as they are at the time the part is created.

The only action of the object factory, `new` creates a new part of the type specified in the `instanceClassName` attribute. It creates this part in memory only; it does not show it. The newly created part is pointed to by the `instance` attribute of the object factory. When this feature is torn off, it is a variable part pointing to the part you just created. By acting on this variable part, you can act upon the part itself. To make the part visible to the end user you can perform the `openWidget` action on the part if it is a GUI application with a window as its primary part. "Adding Parts at Runtime" in topic 2.2.2.3.1 describes techniques for making other parts visible to the end user.

GUI PROGRAMMING EXAMPLE: USING THE OBJECT FACTORY TO OPEN WINDOWS

This example uses the object factory to open multiple windows.

1. Create a GUI application with an entry field on the window. Resize and reposition the window on the free-form surface and then save the GUI application as **OFWIN**.
2. Create another GUI application with a push button called "Open" on the window.
3. Add the **OFWIN** GUI application to the free-form surface. Connect the `clicked` event of the push button to the `openWidget` action of the external GUI application.
4. Save the application as **OPENWIN1**.
5. Test the application.

Every time you click on the push button, the same copy of the window with the entry field is shown. Let's modify the GUI application to open new instances of the entry field GUI application.

1. Put an object factory on the free-form surface of the **OPENWIN1** GUI application. Set the settings and type **OFWIN** in the `instanceClassName` entry field.
2. Tear off the `instance` attribute of the object factory.
3. Delete the **OFWIN** external GUI application from the free-form surface.
4. Connect the `clicked` event of the push button to the `new` action of the object factory. Connect the `clicked` event of the push button to the `openWidget` action of the torn off `instance` attribute of the object factory. Make sure the `new` action is performed before the `openWidget` action.
5. Save the application as **OPENWIN2**.
6. Test the application.

Click on the push button several times. You will notice that each time you click on the push button a new window is created. Type some text in an entry field of one of the windows. If this influences the contents of the other windows. The windows are completely independent of each other.

Every time you perform the new action, a new part is created and the instance attribute contains a pointer to the last created part. If you do not store the pointers to the previously created parts somewhere, you will never be able to access them again. You can store the pointers to the parts in an ordered collection by making a connection to the add: action of the ordered collection and giving it the instance attribute of the object factory as its parameter.

Accessing the part in the ordered collection requires that you know the index of the part in the ordered collection. You can use the atIndex: action of the ordered collection to determine the pointer to the part. Connect the result of this connection to a variable part, and you are ready to use the part by interacting with the variable part.

GUI PROGRAMMING EXAMPLE: IMPLEMENTING CLOSE WINDOW SUPPORT WITH AN OBJECT FACTORY

In this example we add function to close a certain application in the order of opened applications.

1. Start with the **OPENWIN2** application.
2. Save the application as **OPENWIN3**.
3. Add an ordered collection to the free-form surface of the GUI application.
4. Connect the clicked event of the push button with the add: action of the ordered collection. Provide the instance attribute of the object factory as the anObject parameter for the connection. Use the **Reorder connections from** push button Object menu option to move the add: action is performed after the GUI application has been newly created but before the window is opened.
5. Add an entry field to the window and a second push button called "Close." The idea is that clicking on **Close** will close the window number as stated in the entry field.

Note: You can change the entry field data type to Integer or Number, so it matches the type requirements of the target parameter (we have not made this connection yet), but this was not required in our testing.
6. Add a variable part to the free-form surface.
7. Connect the clicked event of the **Close** push button to the atIndex: action of the ordered collection. Provide object of the entry field as the anIndex parameter for the connection. Connect the result of the connection to the self attribute of the variable part.
8. Connect the clicked event of the **Close** push button to the **unlisted action...** action of the variable part. Type in closeWidget as the action to be taken and click on **OK**.
9. Connect the clicked event of the **Close** push button to the removeAtIndex: action of the ordered collection. Provide object of the entry field as the parameter. This ensures that if the ordered collection does not contain parts that no longer exist. All the open windows move one down in the hierarchy if they were opened after the window that was just closed.
10. Save and test the application.

Open a couple of windows and see whether you can close a specific window and open some other windows.

Note: You may trigger a walkback if you use an index value that does not exist in the ordered collection. You cannot close a window that does not exist.

2.2.2.2 Difference between Open and Create

When a GUI application is executed or used, two distinct parts are loaded into memory: the application data and the data associated with this instance of the GUI application. When a next instance of the same GUI application is opened, as can be achieved using an object factory, only the instance data for this part is created in memory. The application data is already there and does not have to be loaded.

In 3GL language terms, you would say that the application data is the code segment and the associated data is the data segment. The same code segment can be used to process different data segments.

A GUI application is loaded into memory when:

- ☐ It is started in the Interactive Test Facility or at runtime
 - ☐ The openWidget action is performed and the GUI application had not been opened yet
 - ☐ The GUI application of which it is part (in the case of embeddable part) is created
 - ☐ A new instance of the part is created using an object factory
- or
- ☐ createPart or backgroundCreatePart action is performed on the part

Once the GUI application has been loaded into memory, it can be communicated with. It does not have to be visible; its data need only be loaded into memory. For some attributes of the GUI application to have an effect, they must be set before the application opens.

+-----+ <u>HINT</u> +-----+	
+-----+	
	When it is relevant that data is passed after the part has been created but before it
	it is best to use an event-to-attribute connection to change the data in the part. T
	connection must fire before the widget is opened but after it has been created.
+-----+	

After a GUI application has been used, it is typically closed. This closing causes both the definition and the data to be kept in memory. This is useful from a caching perspective but also takes up resources if the data is not going to be used again. If there are attribute-to-attribute connections to the data, these connections still exist, and the associated data will be synchronized as changes occur in the system. This can cause tremendous overhead.

The destroyPart action causes the data to be removed from memory. This also removes the associated attribute-to-attribute connections. The definition of a GUI application cannot be removed from memory once it has been created, it stays in memory until the VisualAge Generator GUI application runtime environment is shut down.

+-----+ <u>GUI PROGRAMMING EXAMPLE: USING DESTROYPART TO MANAGE DATA BUFFERS</u> +-----+	
+-----+	
	This example shows that the data associated with a GUI application will not be deleted
	you use the <u>destroyPart</u> action.
	1. Start with the OPENWIN2 application (created as part of GUI programming example U
	Object Factory to Open Windows in topic 2.2.2.1).
	2. Test the application. Open the window by clicking on the push button. Type text
	entry field. Close the window that is shown, using the system menu. Now open t
	again by clicking on the push button. The text will still be there.
	3. Connect the <u>closedWidget</u> event of the OFWIN GUI application on the free-form surf
	<u>destroyPart</u> action of the same OFWIN GUI application.
	4. Save the application as OPENWIN4 .
	5. Test the application again and see whether the previous behavior still exists.

	An alternative to the connection to the <u>destroyPart</u> action is the <u>destroyOnClose</u> attribute setting for the external GUI application on the free-form surface. An external GUI application has settings with respect to the GUI application free-form surface where it has been defined. Currently the only available setting is the <u>destroyOnClose</u> attribute. The setting is used for this instance of the external GUI application on the free-form surface; it does not affect other external GUI application instances.
--	---

2.2.2.3 Putting Parts on a Window

In this section we discuss how a part can be placed on the screen or within another part, such as a window, at runtime, and how you can change the location of the new part on the screen or within the window.

Subtopics

2.2.2.3.1 Adding Parts at Runtime

2.2.2.3.2 Part Placement

2.2.2.3.1 Adding Parts at Runtime

When a part is contained within another part, it has a parent-child relationship to that part. Therefore when you close the parent, the child is automatically closed. Container parts within VisualAge Generator can act as parents.

The following actions allow you to add and remove parts from a parent part at runtime:

removeSubpartNamed:

This action removes the named part from the part at which the action is directed. The result attribute of the connection will contain a pointer to the part. The part is removed from the parent of which it is part but will still exist in memory. (8)

subpartNamed:

This action provides you with a pointer to the part of which the name is provided. By connecting the result to a variable part the part can be acted upon.

subpartNamed:put:

This action will put a part on the containing part at which it is directed and provide it with the given name. The second argument is a pointer to the part, in other words its self. The part will be made part of the container but will not be visible. An openWidget must be issued on the part to make it visible.

subpartNamed:put:beforePartNamed:

This action is similar to the subpartNamed:put: action except that the part is added before the part that is named in the component sequence. The initial component sequence can be viewed by looking at the parts list of a window. The component sequence mainly has an effect on the sequence in which parts are created and destroyed. This setting is only guaranteed if you first close and then reopen the window.

subpartNamed:putOpened:

This action is similar to the subpartNamed:put: action except that the part added will also be displayable (visible) in the target container.

subpartNamed:putOpened:beforePartNamed:

This action is a combination of the previously defined actions. The part is shown and opened at a certain location in the component sequence.

A part in VisualAge Generator has a name. This name must be unique. If you do not give a unique name, the system will give it a unique name or make the name you provided unique by adding a number.

Dynamically adding a part means that its definition should already have been created. At runtime you decide to add it as a visible part. You can of course also have created the part at runtime by using the object factory.

GUI PROGRAMMING EXAMPLE: BUILDING DYNAMIC MENUS

This example shows how to add a series of items to a menu.

1. Create a new GUI application. Add a menu bar to the window with one popup menu item. Change the name of this item to "List."
2. Add an object factory to the free-form surface of the GUI application. Set "AbtPushButtonView" as its instanceClassName. Tear off the instance attribute of the factory.
3. Add a push button called **Add menu item** to the window.
4. Connect the clicked event of the push button to the new action of the object factory.
5. Connect the clicked event of the push button to the subpartNamed:putOpened: action of the popup menu part on the free-form surface. Connect the self attribute of the instance of the object factory to the part attribute of the connection.
6. Add an entry field to the window.

Note: You can change the entry field data type to Integer or Number, so it matches the menu item.

	type requirements of the target parameter (we have not made this connection yet), was not required in our testing.
	7. Connect the <u>object</u> of the entry field to the <u>name</u> attribute of the connection between the push button and the pop-up menu.
	8. Save the application as DYMENU .
	9. Test the application. Every time you click on the push button, an item should be added to the menu with the name as provided in the entry field.
	10. Try changing the push button <u>clicked</u> connection from <u>subpartNamed:putOpened:</u> to <u>subpartNamed:put:</u> . Test the application. What happens?

	<u>HINT</u>
	For more information about and guidance on using dynamic programming functions, review the samples provided with VisualAge Generator. Look for these files in the \SAMPLES subdirectory where you installed VisualAge Generator:
	PARTADR.ESF PARTADR.TXT

(8) The implementation of this action is not complete. The lab may change it, such that there is no result attribute. The implementation may change so that the part you have removed is also destroyed. Experiment as required to ensure that you understand the function provided.

A part is placed within an x,y grid indicated on the part in which it is contained. The origin of the grid has coordinates (0,0). Parts have a width and height indicated in pixels. To change both the x,y position and the height and width, use the corresponding actions of a part and provide the new values as its parameters.

GUI PROGRAMMING EXAMPLE:	MODIFYING PART PLACEMENT IN A WINDOW DYNAMICALLY
	<p>This example shows how runtime placement of parts in a window works.</p> <ol style="list-style-type: none">1. Create a new GUI application. Place four entry fields on the window, labeling the width, and height.2. Put a group box on the window and place a list box inside the group box.3. Add a push button called "Move" to the window.4. Connect the <u>clicked</u> event of the push button to the <u>x, y, width, height</u> actions of the list box and provide the <u>object</u> values of the entry fields as its parameter.5. Save the application as DYNPOS.6. Test the application. Notice that the list box is moved and sized inside the group box because this is the part in which it is contained. <p>The group box is resized with the list box although no relationship has been set between the two parts. This is because the <code>resizePolicy</code> rules, based on the current settings, cause the group box to automatically resize to try and accommodate the child parts.</p>

The `initWidgetSize` of the window can also be used to change the initial position and size of the window. This attribute must be set before the window is opened but after it has been created. The attribute can be torn off from the window and contains two attributes: `origin` and `corner`. Each attribute can be torn off from the tear-off of `initWidgetSize`. They are points requiring an x and y value. Set the values after you have created the window and before you open it.

Placing parts relative to each other and making them resize automatically is discussed in "Defining Visual Parts inside Other Visual Parts" in topic 1.4.3.

2.2.3 Summary

Embeddable parts are a way of creating reusable components and building applications from those components. Embeddable parts also enable you to hide complexity.

Parts communicate with each other through their interface. The interface contains actions, attributes, and events. Data can be accessed by another part if the data has been passed or if it has been given a reference to the data. In the latter case a variable part is used.

A variable part is a pointer. It points to the part that it will represent with its self attribute. A variable part can point to any part in VisualAge Generator.

Parts can be created at runtime using an object factory. Instances created with the object factory can be managed by using an ordered collection. Parts can be added to a container part through dynamic programming. Their x, y position and width and height allow you to dynamically move them within the container of which they are part.

2.2.4 What You Should Now Be Able To Do

You should now have a basic understanding of the creation and use of embeddable parts. You should be able to:

- ☐ Build embeddable parts.
- ☐ Define the interface of an embeddable part.
- ☐ Consider different strategies for making parts communicate.
- ☐ Use a variable part as a pointer to parts elsewhere in the system.
- ☐ Create parts dynamically and if required show them within a container.

2.3 Chapter 12. GUI Error Handling

This chapter discusses how you can handle errors in GUI applications. It looks at the most common types of errors and explains how to present errors to users.

We distinguish three types of errors:

Middleware and server database errors

The direct implementation of middleware and server database errors is part of the server programming domain. Within the context of this book, we only discuss their relationship to GUI applications.

Data validation errors

Data validation errors are discussed separately because validation can occur on both the server side and the client side of an application. Handling the errors and validating the correctness of the data are discussed.

GUI errors

Connections can cause errors, such as when you try to index into an array with an invalid value. We look at ways of preventing errors in connections from occurring.

In the sections that follow we discuss these errors and explain how to inform users about the occurrence of errors.

Subtopics

2.3.1 Middleware and Server Database Errors

2.3.2 Data Validation Errors

2.3.3 GUI Errors

2.3.4 Presenting Error Messages

2.3.5 Summary

2.3.6 What You Should Now Be Able to Do

2.3.1 Middleware and Server Database Errors

VisualAge Generator calls to server applications and server access to database resources can trigger errors. In this section we discuss these errors and explain how you can capture information and control subsequent processing when an error has occurred. Application systems should also implement support for functional errors, but we do not discuss these errors in this section.

Subtopics

2.3.1.1 Middleware Errors

2.3.1.2 Server Database Errors

2.3.1.1 Middleware Errors

VisualAge Generator allows you to check whether an error has occurred during execution of a server application. The error can be caused by:

- ☐ An abend in the server program
- ☐ A problem anywhere between the client and the server

To make sure the control returns to the GUI application after the error has occurred, add the (REPLY option to the call to the server application, as in this example:

```
CALL SRVAPP PARM1, PARM2 (REPLY;
```

Information about the error is provided in the EZERT8 system variable. To test the contents of EZERT8 from a server application, you first have to call a server application from a process or statement group instead of putting the application on the free-form surface and using the execute action of the application. After the call to the server application you can test EZERT8. A nonzero value indicates that an error has occurred. For an explanation of the code values, see *VisualAge Generator Messages and Codes Manual*.

The following sample code checks for the value of EZERT8 after a call to a process:

```
CALL SRVAPP PARM1, PARM2 (REPLY;  
IF EZERT8 NE '00000000';  
  /* Call process to translate EZERT8 code to message code  
  PERFORM RBERZPG-CHK-EZERT8  
END;
```

2.3.1.2 Server Database Errors

A call to a server that accesses a database resource can cause the database to return several indicators about the state of the database. These indicators differ among the different database systems.

For relational database systems, the SQLCA variable contains information about the state of the database system. SQLCA is a structured variable containing a number of indicators.

There are two kinds of server database errors:

Soft errors

Soft errors do not cause the application toabend but indicate a state that is recoverable. For instance the database returns a No Record Found (NRF) indicator when no value was found. If a value was expected, this is apparently an error. If a value was possibly expected, it is no error.

Hard errors

Hard errors usually cause the application toabend. To prevent hard errors, set the VisualAge Generator EZEFE variable to 1 as part of your initialization process.

After each I/O process, you will have to check for errors and determine which soft errors are allowed and which are not allowed. You can implement the checking as a standard error routine. The error routine should also save the relevant error indicators to a working storage record that is passed back to the GUI application because the value of the error indicator, such as SQLCA, is lost when the application is left or another SQL statement or function is executed.

2.3.2 Data Validation Errors

Ensuring that data is correct and represented correctly is an important part of application development. We distinguish three elements:

- ☐ Masking--preventing users from making errors in entering data
- ☐ Validation--checking the correctness of the data
- ☐ Formatting--displaying the data in a correct format

Subtopics

2.3.2.1 Masking

2.3.2.2 Validation

2.3.2.3 Formatting

2.3.2.1 Masking

It is always better to prevent users from making errors rather than fixing the consequences once errors have been made. This is an important part of user interface design.

With VisualAge Generator you can use parts that require the user to make a choice, such as radio buttons and list boxes. At runtime you can set the items attribute of these parts with the valid values. The optimal way of retrieving lookup data is discussed in more detail in "Cache, Cache, Cache" in topic 2.4.4.2.1.

VisualAge Generator also includes a part called *Formatted Text* in the Data Entry category. This part is an entry field with an additional feature. It allows you to give it an edit mask. An edit mask restricts the user from typing in anything other than those elements that correspond to the definition.

The mask is stated as a string; the different characters in the string indicate the type of character expected at a certain point.

You can also add literals to the string by enclosing them in quotation marks. For instance, a U.S. telephone number would be: `"(999)"-999-9999`. VisualAge Generator even provides a number of standard edit masks.

GUI PROGRAMMING EXAMPLE: VALIDATION USING A MASK	
	In this example we create a masked field for a credit card number and expiration date user.
	1. Create a new GUI application. Put a Formatted Text part and a normal entry field window.
	2. Open the settings of the Formatted Text part. Change the Edit mask type to String on the Customize... push button.
	3. Enter 9999B99999B9999B99"/"99 as the format string (see Figure 52) and save the settings.
	4. Connect the <u>object</u> attribute of the Formatted Text part with the <u>object</u> attribute of the entry field.
	5. Test the application (save as member VALMASK). After you have entered data in the Formatted Text part, tab out, and see the results in the entry field. What happens if you enter data in the entry field? What about entering data that does not match the Formatted Text part?

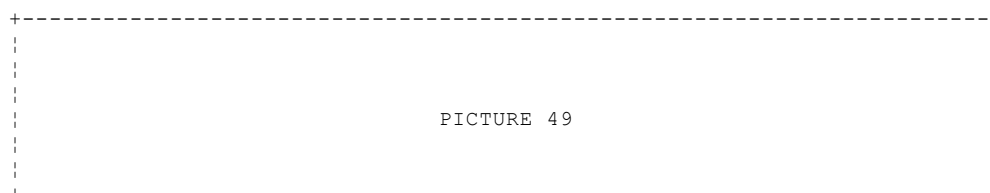


Figure 52. Credit Card Edit Mask

2.3.2.2 Validation

The purpose of validation is to ensure that a value given to an attribute is correct within the business context. Every value can be described as consisting of a number of validating elements: the data type, the collection of valid values, and the dependency on values of other business data. (9)

When implementing validation in an application, consider these user interface design and technical design issues:

- ☐ When the validation should be performed
- ☐ Where the validation should be performed
- ☐ How the validation should be performed

Before we discuss these issues, let us look at the mechanisms VisualAge Generator provides for implementing validation.

- (9) Although a data type is in fact a collection of valid values, we include it for practical reasons.

Subtopics

- 2.3.2.2.1 Converter
- 2.3.2.2.2 Form Input Checker
- 2.3.2.2.3 Using a Possible Values List
- 2.3.2.2.4 Logic
- 2.3.2.2.5 When to Validate
- 2.3.2.2.6 How to Validate

2.3.2.2.1 Converter

Each VisualAge Generator part that contains data can be set to be of a certain data type in its Settings. This setting can be further specified by customizing, for example, the allowable range of values. "Data Format in a GUI Application" in topic 1.7.8 contains a more detailed explanation of the different settings of data types.

The setting is stored in the converter attribute of the part. The converter, like the iterator explained in "Iterator" in topic 1.9.3.1, is a part in itself with its own behavior. You can use the part by tearing off the converter attribute from the part that contains it.

The converter has the following features: (10)

□ Actions

- (none)

□ Attributes

- acceptsDBString - allows double-byte characters to be entered into the field
- caseConversion - indicates whether the text should be converted (possible values are: asIs).
- caseRule - indicates whether the text should be converted to upper case (upper), converted to lower case (lower), or not be converted at all (empty value).
- defaultObjectFromEmptyString - When the part does not contain text, this is the string that is used as the default if the emptyMakesDefault is set to true.
- emptyMakesDefault - indicates whether the value should be set to the default value when the part does not contain text
- justification - value indicates the justification (left, center, or right) of the field in a table part
- max - the maximum allowed value
- min - the minimum allowed value
- minCharsRequired - the minimum number of characters that must be entered in the field.
- name - No meaning was discovered for this attribute.
- worksWellWithContinousNotification - read-only attribute indicating whether this converter works with the **Notify change on each keystroke** setting of the part (for example, False for integer values)

□ Events

The list of events is similar to the list of attributes. The events are the changes that occur to the attributes.

The values for the different attributes of the converter can be set at runtime. (11) Change the converter of one part, for example, on the free-form surface, and assign the converter of this part to the converter of a part on a window. Use an event-to-attribute connection to make sure the value gets passed.

+-----
| GUI PROGRAMMING EXAMPLE: CHANGING A CONVERTER
+-----

|
|
|

+-----
| This example shows how to change the converter of a part at runtime.

- | 1. Create a new GUI application.
- | 2. Put an entry field on the window and a label on the free-form surface.

|
|

We will change the converter of the entry field on the window by changing the con
the label on the free-form surface and copying the settings to the entry field on

	<p>window.</p> <ol style="list-style-type: none"> 3. Change the data type of the label on the free-form surface to "string" and customize its characteristics. 4. Tear off the <u>converter</u> attribute of the label. 5. Create a quick form on the window of the <u>self</u> attribute of the torn-off <u>converter</u> (you may want to size the window part to be taller first). 6. Add a push button to the window and connect the <u>clicked</u> event of the push button <u>converter</u> attribute of the entry field. 7. Provide the <u>self</u> attribute of the torn-off <u>converter</u> attribute as the parameter to the connection. 8. Test the application (save the member as VALCONV). <p>Note the reaction when you type new data in the entry field and tab out. Try changing the values of the converter and click on the push button. See whether there is a change in the value of the entry field. Repeat the process of typing data in the entry field and tabbing out.</p>
--	--

<u>HINT</u>	
	<p>The attributes of the <u>converter</u> for an entry field only contain a subset of the attributes of the <u>converter</u> of a label. Additional values can be set for an entry field, however. Use the <u>converter</u> of a label to set the values. Use an event-to-attribute connection to connect the value of the <u>converter</u> attribute of the entry field. Use the self of the <u>converter</u> attribute of the label as the parameter to this connection.</p> <p>You could also use the unlisted features for the torn-off converter for the entry field to access the missing string data type attributes.</p>

Using the converter causes the field to show the string "***error**" if the value does not conform to the converter and has lost focus. The object attribute of the field will contain the original value, that is, the value before it was changed to an invalid value. If you do not want to use the converter, set the data type of the part to "(none)."

Using the converter causes the userInputConvertError event to be signaled when the user types in an incorrect value instead of the object and string events being signaled. See "Editing an Entry Field" in topic 1.9.1.1.1 for an explanation of the sequence of events when changing the value in an entry field.

(10) The exact list of features shown for a converter depends on the part from which it is torn off. An entry field, for instance, lists a subset of the mentioned features. Some features not listed for the entry field part converter, but shown in our list, may actually exist. For example, an entry field with a data type of string would have the min and max attributes, among others. These can be accessed by using an unlisted attribute.

(11) Although Formatted Text also has a format attribute that contains the way it looks and is validated, its value cannot be changed at runtime. But, there is an unlisted attribute, formatString, which is the mask for the Formatted Text part.

2.3.2.2.2 Form Input Checker

The Form Input Checker uses the converter attribute of a part to determine whether the value it has been given is valid. It looks at all of the fields that are part of its verificationRoot and checks whether each of them converts correctly. If they do not convert correctly, the checkFailed event is signaled. If they do convert correctly, the checkSucceeded event is signaled. Both events can be used to take further action.

The verificationRoot of the Form Input Checker can be connected to the self attribute of any part with a converter or any part that contains other parts, such as a window or form. In this case the Form Input Checker checks all fields that are part of the composite part.

Because the functioning of the Form Input Checker depends on the availability of the converter attribute in a part, the Form Input Checker cannot directly check the validity of data items in working storage records and VisualAge Generator tables.

GUI PROGRAMMING EXAMPLE: USING A FORM INPUT CHECKER

This example shows how to use the Form Input Checker to check the correctness of the an entry field and turn the background color of the entry field red if an error is signaled.

1. Create a new GUI application.
2. Add two entry fields to the window.
3. Set the data type of the first entry field to Integer.
4. Put a Form Input Checker on the free-form surface.
5. Connect the verificationRoot attribute of the Form Input Checker to the self attribute of the first entry field.
6. Connect the losingFocus event of the first entry field to the check action of the Form Input Checker. Using losingFocus ensures that the check always occurs when the user tabs out of the entry field. The object, string, and userInputConvertError events get signaled when the user has changed the data.
7. Connect the checkFailed event of the Form Input Checker to the backgroundColor attribute of the first entry field. Provide "Red" as its parameter (either hard coded or by connecting to the value of a label as a parameter). Also connect the checkSucceeded event of the Form Input Checker to the backgroundColor attribute of the first entry field and provide an empty value as its parameter (this gives the entry field its default color again).
8. Connect the checkFailed event of the Form Input Checker to the setFocus action of the first entry field. This ensures that you are not allowed to leave the field until you have correctly entered the data.
9. Test the application (save the member as **VALFCHK**).

The entry field should turn red when you provide an incorrect value and turn back to its default color when you provide a correct value.

The Form Input Checker generates a message when an error occurs. The message is contained in the errorMessage attribute. The message has the following format:

```
EZE5048E The x digit is not valid.
```

To show the error message to the user, connect the checkFailed event of the Form Input Checker to the showErrorToUser of the same Form Input Checker.

Note: Use caution with the checkFailed-to-showErrorToUser connection. When we defined it during our testing, using VisualAge Generator V2.2 (FixPak 1), we had to use ALT+Print Screen, to break out of a visual loop if we triggered the check attribute of the Form Input Checker using an entry field event such as losing focus. The showErrorToUser action works well when an independent event, such as push button clicked, is used to trigger the check action of the Form Input Checker.

The errorView attribute of the Form Input Checker contains a pointer to the part that was in error. This attribute can be connected to a variable

part for more generic error handling.

2.3.2.2.3 Using a Possible Values List

If one of the elements of validation is checking whether the value falls within a certain collection of allowable values, a list with allowable values could be used to check the validity of a value. The collection should have a limited and definable number of elements. You can use different types of lists for this purpose, depending on your technical environment. The optimal way of retrieving this reference data is discussed in more detail in Chapter 13, "Performance" in topic 2.4.

The solutions provided for checking against a list of possible values can be combined with other solutions.

A VisualAge Generator Table: A VisualAge Generator table can be used to contain all valid values for certain business data. With a VisualAge Generator table, you can check the validity of a value, using the FIND statement in a process or statement group. FIND enables you to take an action when you either find or do not find a value in a VisualAge Generator table. The FIND statement has the following syntax:

```
FIND DATAITEM TABLE.SEARCHCOLUMN ACTION-IF-FOUND ACTION-IF-NOTFOUND;
```

When you do not find a value, one possible action is to use a process that sets up the display of an error message. One of the advantages of using a VisualAge Generator table is that it only has to be defined once and resides in memory to be shared by all GUI applications in which it has been embedded (see "VisualAge Generator Tables" in topic 1.7.5 for a further explanation of VisualAge Generator tables).

| GUI PROGRAMMING EXAMPLE: VALIDATING INPUT WITH VISUALAGE GENERATOR TABLES

This example shows the use of a VisualAge Generator table to check for the valid entry of airport codes. The table contains two columns and could also be used for populating a list. Often, though, users are happier with typing in the codes they are used

1. Create a VisualAge Generator table, **AIRPORT**, with the following structure:

```
10 AIRPORTCODE      char 3
10 AIRPORTNAME      char 50
```

2. Populate the table with some airports you are familiar with, use upper case letters for the airport codes.
3. Create a new GUI application. Add two entry fields to the window. Customize the first entry field to ensure that three characters are entered and Uppercase conversion is on for the input.
4. Create a working storage record, **CHECKWS**, with the following structure:

```
10 AIRPORTCODE      char 3
10 AIRPORTNAME      char 50
77 CHECKSUCCEDED    char 1
77 CHECKFAILED      char 1
```

The working storage record will be used as a trigger that causes the entry field to be cleared when an incorrect value has been entered. It will also contain the value the user entered for the airport code and return the airport name that matches the code.

5. Connect the first entry field to the AIRPORTCODE data attribute of the working storage record. Connect the second entry field to the AIRPORTNAME data attribute of the working storage record.
6. Create a process, **CHECK**, that will check whether the airport code entered by the user is correct. The process contains the following statements:

```
FIND CHECKWS.AIRPORTCODE AIRPORT.AIRPORTCODE CHECKSUCC CHECKFAIL;
```

7. Save the **CHECK** process member.
8. Select **Define** and **Structure list...** options from the VisualAge Generator Developer menu active while editing the **CHECK** process.

This will show you the statement groups you need to define.

.
. .
.

	<p>.</p> <p>.</p> <p>.</p> <p>9. The CHECKSUCC statement group should contain the following logic:</p> <pre>CHECKWS.CHECKSUCCEEDED = "Y"; MOVE AIRPORT.AIRPORTNAME(EZETST) TO CHECKWS.AIRPORTNAME;</pre> <p>10. The CHECKFAIL statement group should contain the following logic:</p> <pre>CHECKWS.CHECKFAILED = "Y"; MOVE ' ' TO CHECKWS.AIRPORTNAME;</pre> <p>11. Connect the <u>losingFocus</u> event for the first entry field to the <u>execute</u> action of process.</p> <p>12. Connect the <u>CHECKFAILED</u> data event of the working storage record to the <u>background</u> attribute of the first entry field. Provide "Red" as its parameter.</p> <p>13. Connect the <u>CHECKFAILED</u> data event of the working storage record to the <u>setFocus</u> attribute of the first entry field.</p> <p>14. Connect the <u>CHECKSUCCEEDED</u> data event of the working storage record to the <u>background</u> attribute of the first entry field. Provide an empty parameter value.</p> <p>15. Test the application (save the member as VALTAB).</p>
--	--

An Ordered Collection: You can also use an ordered collection with values to check the validity of a certain entry. The ordered collection can be populated from a working storage record, a list box, or any other array of data.

You can check whether a value is in the ordered collection by triggering the indexOf action and providing the connection with the value you want to check for. If the value is found in the ordered collection, the result attribute of the connection will contain a nonzero value. You can check this value in a process or statement group similar to the check performed when using a VisualAge Generator table.

2.3.2.2.4 Logic

Validation can be performed in processes or statement groups to see if data items contain valid values. Therefore the data must always be moved to data items first. Once the data is available, the process can check whether it is valid. Using logic to check the data can be combined with other solutions.

Defining the validating processes requires that you define the rules to which the data elements in your business should abide and building the rules into:

- VisualAge Generator algorithms without performing I/O to a database or file. VisualAge Generator provides a number of possibilities to check whether data items are numeric, blank, or null.
- VisualAge Generator algorithms that use the functionality of the database system to check the validity of the data, for example, to check for valid date values or referential data
- Calling external functions that you may still have available from legacy systems

If an error occurs, initialize the display of an error message to the user can by using a triggering data item in a working storage record. See the example of using a VisualAge Generator table to validate data.

2.3.2.2.5 When to Validate

When to validate is first of all dictated by the preferred behavior of the application toward the user. From this perspective three requirements can be defined:

- ☐ The user should be warned of a possible error as soon as possible.
- ☐ The manner in which the data is validated should be consistent.

For example, when a user has corrected a value and has gone on to correct other values, they should not later be told that the value is still incorrect. Although it may be a technically optimal solution to first check on the client side of the application, and only when the user starts an action check on the server side, this is not consistent from a user perspective.

- ☐ The user should not be allowed to continue unless the error is corrected if the error could cause inconsistent data in your application.

A number of events can be used to cause validation of the data entered by the user to be performed:

- ☐ On each keystroke
- ☐ If you perform an action that requires the data to have been validated
- ☐ When an entry field loses focus

Keystroke: Using each keystroke to check whether data has been changed causes a lot of additional actions to be performed throughout the system, especially when a checking mechanism is used that is not very quick. For example, going to the server to check against the database if a value is valid on each keystroke is probably not a very good idea.

This option does allow you to conform to the requirements from a user interface perspective.

Action: Validating data only when the user performs an action that requires the data to have been validated enables you to do the checking on both the client side and the server side. You have to check on both sides before the user is given feedback on the result to prevent a data item from being validated as correct on the client side and incorrect at the server side. This causes inconsistent behavior of the application to the end user.

Another issue is that it is difficult to provide clear feedback on multiple compounding errors.

Losing Focus: Validating only when a field loses focus is a middle-of-the-road solution. Although it provides a consistent interface, sometimes you have to do a server call to check the data, slowing down response time and increasing the load on the server system.

If you use both the message box of the Form Input Checker and the losingFocus event to show errors to the user, you will encounter problems. Because the opening of a window is an action that causes asynchronous events, if you use the setFocus action to put the control back on a field that was incorrect, the field will get focus. However, because another window is modal on top of it, it will directly lose focus again, causing an endless loop of errors to occur.

To circumvent this behavior, use a flag that indicates whether the window or the field has actually lost focus. Set the flag on the losingFocus event of the window. Do not validate while the flag is set. Clear the flag on the gettingFocus event of the window. Make sure validation is performed when the flag is not set.

If losingFocus is used to validate the data in an entry field and GUI application processing the requires that complete validation is implemented in the GUI application by using a menu option, you have to force the current field to lose focus to be sure you get the latest correct data. A menu does not get the focus in a GUI application, so the entry field will not lose the focus.

Given these three options and their consequences, checking the data on losing focus is the most viable solution if you can circumvent using the Form Input Checker message box to indicate that an error has occurred.

```
+--- Validation: An Alternative View -----+
|
| Choosing an approach to validation is one of the more difficult tasks
| when developing a GUI application system. In many respects the choice
| is not based on the technology used or techniques permitted but
| instead on the value and function of the approach chosen to the user
| community. Good validation that is painful to the users (constant
| reminders of bad input, task interruption, frustration at lack of
| control) is counterproductive.
|
| Do not ignore a simple option: Use a Validate push button on all
| input (data entry window) GUI applications. This allows users to
| decide when they want their input proofread. This could be at the end
| or part way through the data entry process; their choice. You would
| also trigger this validation logic prior to a save request. Very
| flexible, and maybe easier to implement!
|
+-----+
```

2.3.2.2.6 How to Validate

If you decide to validate the data when a field loses focus, there are a number of considerations to be made concerning the implementation of the validation. First of all you have to decide which, or which combination of, mechanisms to use. You also have to decide what to do about some special circumstances, such as cross-field dependencies.

Validation Mechanism: The choice for a preferred validation mechanism is determined by a number of factors:

Reusability

The reusability of the validation across different platforms and between TUI and GUI applications.

Ease of implementation

If it is easier to build and maintain a specific instance of validating logic instead of building reusable parts then there is no reason for reusable validation mechanisms.

Required data

If you need data from the database to be able to perform a check, checking in logic is the only option (if it is not possible to build up a list of possible values).

Check on losing focus

The solution should be in accordance with the standard set for the user interface interaction for validation.

The first observation is that business data with the same validating requirements are used in multiple places throughout the application. This leads to the observation that the implementation of the check should be independent of the location where the check is used. Within a VisualAge Generator GUI application this would mean the validation should be implemented as a separate part.

Within this part you can select one or a combination of mechanisms that best suit the needs for that specific business data element. The combination could be a Form Input Checker, a list of possible values, and a check on the database. These can be sequenced so that you first perform checks on the client; only when those checks have been successfully performed do you check the data item against the server validation. This is also an advantage of checking the data when a field loses focus instead of upon an action.

Implementing this requires that you be able to generically reference a field. To generically reference a field, use a variable part. A variable part points to nothing and does not have any function until you connect it to the self attribute of a part. It then points to the part, and you can perform actions on it just like it is the part. Thus you could also do validation on the variable part if you connect it to a field.

GUI PROGRAMMING EXAMPLE: CREATING A REUSABLE BUSINESS DATA ENTRY FIELD

This example shows the implementation of a generic part for a business data element with the following characteristics:

- ☐ The value should be an integer value.
- ☐ It should be between 222222222 and 777777777.
- ☐ The sum of the individual digits should be divisible by 11.

Here is how to implement such a part.

1. Create a new GUI application. Since the embeddable part for validating is not yet visible, replace the window with a label. We use a label to indicate the name of the part when it is not visible and will only be used on the free-form surface. Give the label the text "digitCheck."
2. Place a variable part on the free-form surface and change its name to **digit**. Provide the self attribute of the variable part as digit.
3. Now we have to provide the part with the correct converter. Put a label on the free-form surface, give it the integer data type. Customize the label so that it has minimum and maximum values of 222222222 and 777777777. Provide a default value of 23456789. Connect the converter attribute of this label part (you will have to carefully resize the label part first).

4. Make a connection from the variable part to the variable part (yes, to itself), by aboutToOpenWidget unlisted event to the converter unlisted attribute. Connect the attribute of the converter as a parameter to the previous connection. You first connect it to the result attribute of the connection then edit the connection and result text into value.

This will make sure the converter is copied to the field that is represented by the variable part in this embedded GUI application.

.

.

5. Put a Form Input Checker on the free-form surface. Connect the verificationRoot to the self attribute of the variable part. Promote the check action of the Form Input Checker as checkDigit. We will control when edit checking is processed.

If the check succeeds we have to check whether the value can be divided by 11. We will do this in a process and require the value to be in a working storage record.

6. Create a working storage record named **VALREUSE-WS** with the following structure on the free-form surface:

```

10 DIGIT                num 10
12 SINGLEDIGIT          num 1   occurs 10
10 TOTAL                num 2
10 REMAINDER            num 2
10 INDEX                num 2
10 ERRORREASON          char 80
77 CHECKFAILED          char 1
77 CHECKSUCCEDED        char 1

```

7. Connect the object attribute of the variable part (and thus of the field) using value attribute...) to the DIGIT data attribute of the working storage record.
8. Create a process named **VALREUSE-CHK** with the logic to check whether the sum of the individual digits can be divided by 11.

```

INDEX = 1;
TOTAL = 0;
WHILE INDEX LE 10;
    TOTAL = TOTAL + SINGLEDIGIT(INDEX);
    INDEX = INDEX + 1;
END;
REMAINDER = TOTAL // 11;
IF REMAINDER EQ 0;
    CHECKSUCCEDED = "Y";
ELSE;
    CHECKFAILED = "Y";
    MOVE "PROC01 Input not divisible by 11." to ERRORREASON;
END;

```

9. Put the process on the free-form surface and execute it when the check of the Form Input Checker succeeds. We still have to show the user that something is wrong.
10. Connect the checkfailed event of the Form Input Checker to the CHECKFAILED data attribute of the working storage record. Provide a parameter value of "Y" for this connection.
11. Connect the CHECKFAILED data event of the working storage record to the background attribute of the variable part (which you have to type in using unlisted attribute the same for the CHECKSUCCEDED data event of the working storage record.

.

.

12. Make two labels. Make the label for one "Red" and the other "Green." Connect "Red" to the result attribute of the connection that occurs from the working storage record.

check fails. Connect the "Green" label to the result attribute of the connection occurs if it succeeded. Change both label-to-connection connections to have value target instead of result by typing the correct value into the settings of the connection.

13. Connect the CHECKFAILED data event of the working storage record to the setFocus attribute of the variable part (use unlisted action...).

Now we have to tell the user what is wrong. There could be two things; input out of range or input not divisible by 11.

14. Connect the Form Input Checker errorMessage attribute to the ERRORREASON data attribute of the working storage record.

15. Promote the ERRORREASON data attribute as errorReason.

16. Save the embedded GUI application as **VALREUSE**

We have now built our validating part. Lets use it in a simple GUI application.

17. Create a new GUI application. Put two entry fields and a push button on the window. Connect the embedded GUI application you created (**VALREUSE**) on the free-form surface.

18. Connect the self attribute of the first entry field to the digit attribute of the embedded GUI application.

19. Name the push button "Validate Window." Connect the clicked event of the push button to the checkDigit action of the embedded GUI application.

20. Make the window part wider and then add a label to the bottom of the window. Set the label (or use the layout settings) to make it as wide as the window part. Better yet, connect the label to the left and right sides and the bottom of the window so they stay together. Make sure the label is tall enough to display a text string.

21. Edit the settings and change the label string to several blanks (general settings page 1), the Alignment to left (general settings page 2), and the Border width to 1 (general settings page 3). This will give us a framed box to put validation messages in during runtime.

22. Connect the errorReason attribute of the embedded GUI application to the object attribute of the label.

23. Test the application (save the member as **VALRTST**) and see if it works as it should. Try input values: 2345678000, 3382810017, and 7758299800.

Make sure your input fails the range and division tests. You should see appropriate messages in the label after you have clicked on the push button.

In this example we show the preferred way of implementing your validation. The complexity is hidden for the developers. You only have to create one validation part for each business element type. The parts are reusable and easily maintained.

Cross-Field Validation: Using a part to check each individual field leaves us still to consider those situations where there is a dependency between the data in different fields. In these cases instead of an individual data type containing some form of logic concerning the valid values, the combination of these fields also determines the validity of the data in an individual field.

The combination of fields could in itself be seen as one field. To this the same would apply as when trying to validate one field. Put the implementation into a separate part if the combination of fields is a recurring theme in the application. The parts for checking the individual fields are part of this aggregate part.

GUI PROGRAMMING EXAMPLE: VALIDATION USING PROCEDURAL LOGIC

In this example we build on the part that checks the validity of a digit that has to be 10 digits and of which the sum of the individual digits must be divisible by 11. We create a window in which this type of data element occurs twice. The first occurrence has to be a smaller digit than the second. We will check the two entry fields in the window separately and then together.

1. Edit the GUI application **VALRTST** and embed a second instance of the digit checking GUI application. Save the GUI application as **VALRTST2**.

2. Connect the self attribute of the second entry field to the digit attribute of the embedded GUI application instance.
3. Change the embedded GUI application to promote the CHECKSUCCEDED data attribute working storage record as checkSucceeded.
4. Save the embeddable part and return to the **VALRTST2** GUI application. From both applications, tear off the digit attribute.
5. Create a working storage record named **VALREUSE-WS2** the following structure:

10	DIGIT1	num	10
12	DIGIT1CHAR	char	10
10	DIGIT2	num	10
12	DIGIT2CHAR	char	10
10	ERRORREASON	char	80
77	CHECKSUCCEDED	char	1
77	CHECKFAILED	char	1
6. Connect the object attribute (use unlisted attribute...) of the tear off of the object attribute of the first embedded GUI application to the DIGIT1 data attribute of the working storage record and the same attribute of the second embedded GUI application to the data attribute of the working storage record.

.

.

.

.

.

.

7. Now we need a process to determine whether the required condition between the two fields has been satisfied. Create a process named **VALREUSE-CHK2** with the following statements to the free-form surface:


```
IF DIGIT1 LE DIGIT2 OR
  DIGIT1CHAR IS BLANKS OR
  DIGIT2CHAR IS BLANKS;
  CHECKSUCCEDED = "Y";
ELSE;
  CHECKFAILED = "Y";
  MOVE "PROC02 First entry not less than second." to ERRORREASON;
END;
```
8. Connect the CHECKFAILED data event of the working storage record to the backgroundColor attribute of both torn-off attributes (using unlisted attribute...).
9. Add a label with the value "Pink" to the free-form surface. Use this label as the background by first connecting the object of the label to the result attribute of the previous connections (CHECKFAILED data to backgroundColor) and then changing the connection from result to value by editing the settings of the connection.
10. Remove the connection between the first embedded GUI application and the label instance used to show error text. Add a connection from the ERRORREASON data attribute of the working storage record to the object attribute of the label.
11. Add a connection between the errorReason attribute for each embedded GUI application to the ERRORREASON data attribute of the working storage record.

This will show any error messages to the user.
12. Connect the checkSucceeded event of the first embedded GUI application to the checkSucceeded event of the second embedded GUI application. Connect the checkSucceeded event of the second embedded GUI application to the execute action of the **VALREUSE-CHK2** process.

This will allow each entry field to be edited in sequence and then the cross edit to be performed.
13. Test the **VALRTST2** application to see whether it works.

You should be able to trigger errors in both entry fields, individually and together, and see the appropriate error message in the label on the window.

HINT

	If a variable part does not have type information associated with it, you can only di promote <u>self</u> . To promote another attribute, you have to first choose Tear off attrib the Object menu of the variable part. Select Other... and type in <u>object</u> . This make tear-off of the <u>object</u> attribute. Now promote <u>self</u> of this tear-off as <u>value</u> .

2.3.2.3 Formatting

Typically, the values that a user enters in a field are formatted according to the converter of the field. If the converter as set in the Settings or at runtime indicates that a digit should be shown with two decimal places, it is shown that way when the users leaves the field.

When the converter has not been defined for a field, you have to create your own converter to ensure that the field shows the data in the format in which the user would like to see it (for example, currency format). Because this is a lot of work, we suggest that you always use the converter. Set it in a reusable part as described in "How to Validate" in topic 2.3.2.2.6.

2.3.3 GUI Errors

Errors in GUI applications can be caused by connections that are incorrect or errors in logic. The handling of the latter we assume is identical to the handling of errors in server applications. Errors in connections are very different, however.

Connections cause errors in the following situations:

- The feature that is used for the connection is not available.

This situation will cause the action, or change in the attribute, to not be performed. It does not cause a runtime error if the feature does not exist in a variable part or a torn-off attribute. If the target feature is part of a real part you will get a runtime error (see "Reading the WALKBACK.LOG" in topic 1.5.2.2).

- The parameter passed to the connection is of the incorrect type.

This situation causes a runtime error.

- The parameter passed to the connection has an invalid value.

This situation causes a runtime error. Whether a value is invalid or not is not always evident. An index value of 0 is invalid for the atIndex: action of an ordered collection but valid for the removeAtIndex: action of an ordered collection.

In the cases in which a runtime error is the result of a connection, the validity of the connection cannot be tested from the connection itself. You have to test the values that are provided to the connection before the connection is triggered. Determine whether or not to trigger the connection depending on the outcome of this test.

The process of checking every parameter to a connection is cumbersome and should be performed only in situations when the error could occur in normal execution of the program. Completely unexpected and undesired behavior during runtime can then still cause a runtime failure of the application, but the failing condition can be logged. Because there are no logical units of work (LUWs) on the client application, the inconsistency of your business data is not endangered.

More dangerous are conditions that are unintentional and do not cause a runtime error. These situations are best covered by a good design resulting in an accurate test plan.

There is a valid argument for wanting connections to provide more information than they do, especially connections made to your own parts that perform an action based on that connection triggering. For example, a part that translated terms from one language into another would have an action translate:. It would be useful if the result attribute of a connection to this action contained the translated term. Currently there is no facility in VisualAge Generator for implementing this kind of behavior.

2.3.4 Presenting Error Messages

Once an error has occurred, the user should be notified in a meaningful way that gives him or her enough information to proceed. The information should not be too technical in nature. That is why you may want to log the error for later reference by a technical support team.

Subtopics

2.3.4.1 Using Messages

2.3.4.2 Message Window

2.3.4.3 Logging Errors

2.3.4.1 Using Messages

Errors are represented to the user as messages. Messages can come from different sources and have different contents. To have some form of control over the messages you should get them from a permanent storage. The error that occurred can be translated to the appropriate message code that relates to the appropriate error message to show the user as stored in the permanent storage. This also reduces the amount of data passed between the server and the client.

An error condition that occurs within the system can provide you with two types of information:

- Generic error type

This indicates the type of error, for example, the fact that you are trying to insert a *record* into the database that already exists. This error may be triggered by the server database error code "DUP."

- Specific error conditions

The term *record* in the generic error type is replaced with a meaningful name, for example, *Customer*.

Subtopics

2.3.4.1.1 Message Table

2.3.4.1.2 Placeholders

2.3.4.1.1 *Message Table*

A message table contains messages corresponding to a certain message code. The code determines the description to be provided to the user when the error as indicated by the message code occurs. The table can be a VisualAge Generator table or a DB2 table. The optimal way of retrieving this message data is discussed in more detail in "Cache, Cache, Cache" in topic 2.4.4.2.1.

2.3.4.1.2 Placeholders

The variable part of the message indicates the specific circumstances in which the error occurred. They are indicated within the message with a special character reserved for the purpose, for instance, %. If an error occurs and both the message code and variable information are provided by the application in error, you will need a process to replace the placeholders with their replacements. You have to find the message and search the string for the placeholders. Then substitute each occurrence of the placeholder with its respective replacement.

GUI PROGRAMMING EXAMPLE: DISPLAYING CUSTOMIZED MESSAGES

In this example we show a technique that formats a specific error message given the message code, values for up to three identified placeholders in the generic error message. The VisualAge Generator table with message code and unformatted message text values.

1. Create two working storage records, **MSGWS1** and **MSGWS2**, with the following structure:

MSGWS1

10	MESSAGECODE	char	3	
10	PLACEHOLDER	char	20	occurs 3
10	BASEPLACEHOLDER	char	20	
12	CHARBASEPLACEHOLDER	char	1	occurs 20

MSGWS2

10	BASEMESSAGE	char	255	
12	CHARBASEMESSAGE	char	1	occurs 255
10	MESSAGE	char	255	
12	CHARMESSAGE	char	1	occurs 255
10	SOURCEINDEX	num	3	
10	PLACEHOLDERINDEX	num	1	
10	INSERTINDEX	num	3	

2. Create a VisualAge Generator table, **MSGTBL** with the following structure and provide contents with some values (make sure they have some placeholders):

10	MESSAGECODE	char	3	
10	MESSAGE	char	198	

.

.

.

.

.

.

3. Create a process (**MSG-PROC**) with the following code which will build the formatted message from the text in the VisualAge Generator table and the provided placeholders. Placeholders in the base message are represented as a single character ("%").

```
/* Lookup messagecode in VisualAge Generator Table
RETR MSGWS1.MESSAGECODE MSGTBL.MESSAGECODE MSGWS2.BASEMESSAGE MESSAGE;
IF EZETST EQ 0;
  /* Message was not found provide default message
  MSGWS1.MESSAGECODE = "DFL";
  RETR MSGWS1.MESSAGECODE MSGTBL.MESSAGECODE MSGWS2.BASEMESSAGE MESSAGE;
END;
/* Init formatted target with source message (required if no place holder
MSGWS2.MESSAGE = MSGWS2.BASEMESSAGE;

/* Init source, placeholder and formatted target index values
SOURCEINDEX = 1;
PLACEHOLDERINDEX = 1;
INSERTINDEX = 1;

/* As long as there are placeholders
WHILE '%' IN MSGWS2.CHARBASEMESSAGE(SOURCEINDEX);
  INSERTINDEX = 1;
  /* Put text up to placeholder in target variable
```

```
MOVEA MSGWS2.CHARBASEMESSAGE TO MSGWS2.CHARMESSAGE(INSERTINDEX) FOR
EZETST;

/* Adjust source and target index values
SOURCEINDEX = EZETST + 1;
INSERTINDEX = EZETST;

/* Find placeholder end (Only supports one word) and load to target
MOVE MSGWS1.PLACEHOLDER(PLACEHOLDERINDEX) TO MSGWS1.BASEPLACEHOLDER;
IF " " IN MSGWS1.CHARBASEPLACEHOLDER;
    MOVEA MSGWS1.CHARBASEPLACEHOLDER(1)
    TO MSGWS2.CHARMESSAGE(INSERTINDEX) FOR EZETST;
    INSERTINDEX = EZETST;
END;

/* Put last part of message in formatted target
IF " " IN MSGWS2.CHARMESSAGE;
    MOVEA MSGWS2.CHARBASEMESSAGE(SOURCEINDEX) TO
    MSGWS2.CHARMESSAGE(INSERTINDEX);
END;
/* Put message back in source variable for next loop iteration
MSGWS2.BASEMESSAGE = MSGWS2.MESSAGE;
PLACEHOLDERINDEX = PLACEHOLDERINDEX + 1;
END;
```

4. Create a GUI application and place the working storage records, the VisualAge Gen table, and the process on the free-form surface. Put four entry fields and a push button in the window.

.

.

.

.

.

.

5. Connect the object attribute of the first entry field to the MESSAGECODE data attribute of the working storage record **MSGWS1**.
6. Tear off the PLACEHOLDER attribute of the **MSGWS1** working storage record.
7. Connect the clicked event of the push button to the setValueAtIndex:using: action of the process. Provide hardcoded indexes for the connections using the number 1, 2, and 3, one per connection.
8. Connect the object attribute of each remaining entry field as the value attribute of the connections between the push button and the torn off the PLACEHOLDER attribute. Provide the entry field order and the indexes hardcoded in the connections.
9. Connect the clicked event of the push button to the execute action of the **MSG-PROC** process.
10. Make the window part wider and then add a label to the bottom of the window. Set the label (or use the layout settings) to make it as wide as the window part. Better yet, connect the label to the left and right sides and the bottom of the window so they are all together. Make sure the label is tall enough to display a text string.
11. Connect the MESSAGE data attribute of the working storage record **MSGWS2** to the object attribute of the label.
12. Test the application (save the member as **MSGFUN**). Enter a message ID in the first entry field, values for any marked placeholders ("%") in the remaining entry fields and click the push button. A formatted message will be displayed in the label.

To show the error in a message window, you could include information on the look and feel (like the type of icon to be displayed) of the message window in your storage of the messages. Although this information will not be used when building TUI applications, there is no harm in including it.

2.3.4.2 Message Window

VisualAge Generator provides different mechanisms for showing error messages to the user. The mechanism you choose depends on the circumstance of the error and the architectural requirements of your application.

Information Area: A part of a window can be allocated as an information area. This part of the window can then be used for messages to the user that are not critical for him or her to respond to but provide feedback on his or her actions. The information area can be created as a label at the bottom of the window. If you attach its sides to the window, it will automatically resize with the window. The label can be set to have a border to clearly distinguish it from the rest of the window.

```
+-----+
|  HINT  |
+-----+
|        |
+-----+
|        | To add a border to a label change the Border width attribute on page 3 of the General
|        | settings notebook.
+-----+
|        |
+-----+
```

Use of an information area is advisable for all messages to which a user does not have to respond. This could even be made a feature that allows different behavior for advanced users and novice users. See the CUA manual for a more detailed description of the information area from a user interface perspective.

Message Window of a Form Input Checker: The `showErrorToUser` action of the Form Input Checker allows you to show the default error message provided by the Form Input Checker in its own message window.

A disadvantage of this technique is that the message cannot be changed. The message window that is shown also allows for just one user action, clicking on the **OK** push button. Because the message window only works if an error has been detected by the Form Input Checker, you will need other message windows for other situations. Building such inconsistency into your application is not advisable. The message provided by the Form Input Checker is also not very user friendly.

Message Action on a GUI Application: Every GUI application has an action called `message:title:iconType:buttonType:` with optionally `helpFile:helpTopic:`. This action is available from the Connect window when you choose **Connect** and then **All features...** from the Object menu of the free-form surface. The action provides the default OS/2 or Windows help dialog box with the message, title, icons, buttons, and help information that you provided as its parameters. For a full explanation of the possible values of these parameters see the *VisualAge Generator GUI User's Guide and Reference*.

The window is shown based on the language defined to the operating system. This source of language control does not allow multiple languages to coexist within your application. The window is also always modal to your entire VisualAge Generator application. Although you cannot deviate from the default-provided button sets and icon types, they are CUA-conforming and you should not deviate in your application from these standards.

Because this message window is not a part like other parts in your application, it is more difficult to interact with.

Message Prompter: The message prompter is a part in the Prompters category of the palette. It provides almost the same function as the message window part of the GUI application. The major differences are:

- ☐ There is no way of providing a help button and indexing into a help file.
- ☐ You are free to determine the modality of the window. The message window can be system modal, modal to your application, or modal to the window that you provide as its parent.
- ☐ Because the message prompter is a part, you have more control over it than you do over the message window.
- ☐ You can choose the default push button.

Own Message Window: Of course you can build your own message window. Building your own message window is the only viable solution if your application has to support multiple languages at runtime.

We advise you to use the message prompter part as your message window for all situations, unless it is possible to use an information area or the support of multiple languages at runtime is of critical importance to your application. Although the current implementation of the message prompter does not provide support for a help push button, this function is expected.

2.3.4.3 Logging Errors

For purposes of tracking problems and supporting users it is helpful to log error messages that have occurred. Logging could be something that is dependent on the state of the system (for example, only during system and acceptance testing).

Logging can be implemented by adding a string to a multiline edit and writing the string out as the buffer of a file accessor part.

A line can be entered at the beginning of a multiline edit by using the action insertTextAtPosition. This action requires two parameters: text and position. If you leave position blank, the string is added to the beginning of the multiline edit. To be sure the text is on a line by itself, put a linefeed character (ASCII code 13) at the end of the string you want to enter. (12)

If you want to put the text at the end of the multi-line edit use a very large number for position. This, as long as the number is larger than the multi-line edit contents, will put the new text at the end.

GUI PROGRAMMING EXAMPLE: WRITING A LOG FILE	
	This example shows how to build up a log and write it to a file.
	1. Create a new GUI application.
	2. Add a push button, an entry field, and a multi-line edit to the window and a file accessor part to the free-form surface.
	3. Connect the <u>clicked</u> event of the push button to the <u>insertTextAtPosition</u> action of the multi-line edit. Provide the <u>object</u> attribute of the entry field as the <u>text</u> parameter of the connection.
	4. Open the settings for the <u>clicked-to-insertTextAtPosition</u> connection and provide 0 for the parameter <u>afterPosition</u> (or a value of 99999 if you want to append to the end of the file).
	5. Put a label on the free-form surface and give it a file name as its text. Connect the <u>object</u> attribute of the label to the <u>fileSpec</u> attribute of the file accessor part.
	6. Connect the <u>buffer</u> attribute of the file accessor part to the <u>object</u> attribute of the multi-line edit.
	7. Connect the <u>clicked</u> event of the push button to the <u>write</u> action of the file accessor part.
	8. Test the application (save as member VALLOG). Enter a message in the entry field and click on the push button. The text is added to the multi-line edit and is written out to the file. Look at the difference if you add ASCII character 13 at the end of the string before adding. (To add ASCII character 13, hold the Alt key and type "0013" on the numeric keypad.)
	Note: With VisualAge Generator V2.2 (FixPak 1) the use of an <u>afterPosition</u> parameter of 0 results in a rewrite of the first byte of the existing data. This was fixed in V2.3.

Another way of implementing logging is to route the message to some kind of routine (for example, a routine written in REXX or C) or pipe it, using operating system commands.

- (12) This can be added generically to your application that builds up the error messages. Performing it only when the EZESYS variable indicates that the application is running in a client environment.

2.3.5 Summary

Errors can occur in your server application, business data, and GUI applications.

Errors in server applications can be checked by using the VisualAge Generator facilities for calling applications such that they return the status of the called application. Any database systems that are accessed in the application will also provide information on their status that can be used to handle exceptions correctly.

Data entry can prevent users from making mistakes, the values can be checked for accurateness, and they can be formatted in such a way that they are understandable to the user.

Restricting data entry can be achieved by using parts like list boxes, radio button sets, and Formatted Text.

Every value can be described as consisting of a number of validating elements: the data type, the collection of valid values, and the dependency on other business data. VisualAge Generator provides a number of mechanisms for validating these elements of data.

The converter is an attribute of every part that has a data type. It determines the valid values for a part. These can be changed at runtime. The Form Input Checker uses this information to determine whether a value has been correctly entered by a user. A list of possible values and logic can also be used to check the correctness of data.

The preferred time to validate data is when a field loses focus. It is best to use a reusable part that corresponds to a certain business element definition as the validating element in your application. If there are cross-field dependencies, these can be taken care of by combining the relevant validating parts.

Formatting of data for the purpose of representation to the end user is a function of the converter.

Errors in connections between parts in GUI applications can only be prevented by checking the values provided to the connection before the triggering of the connection. You should determine in individual cases whether this is necessary. Normally your test cases should take these abnormal cases into account.

Indications of errors consist of two parts: a fixed message and a number of variable portions (placeholders) that are put into the message at predetermined locations. The final message is preferably shown using the message prompter part from the Prompters category of the palette. If the support of multiple languages at runtime is a concern for your application, you need to build your own message window.

Error messages can be logged by adding them to a multi-line edit and writing the multi-line edit out to a file.

2.3.6 What You Should Now Be Able to Do

This chapter should have given you the necessary understanding for implementing techniques that ensure the correctness and consistency of your data in your applications. By now you should be able to:

- ☐ Understand the different types of errors and know how to handle each one of them.
- ☐ Understand the usage of (REPLY, EZERT8, and EZEFEFEC).
- ☐ Know the difference between hard and soft errors and the implication for your application design.
- ☐ Define the elements that are of importance in ensuring that your system only contains correct data and that the data is represented in a correct manner to the user.
- ☐ Be able to use masks and other restricted forms of data entry.
- ☐ Define the basic elements of data that can restrict the possible values.
- ☐ Use and understand the converter, Form Input Checker, lists of possible values, and logic to check the validity of entries.
- ☐ Understand and be able to apply the preferred way to implement validation, concerning both the moment of validation and the techniques used for validation.
- ☐ Have data throughout your application formatted to the user's requirements.
- ☐ Determine whether it is necessary to check the validity of a connection in a GUI application.
- ☐ Be able to build a structure for the provision of error messages based on a fixed message portion combined with placeholders.
- ☐ Show error messages using the message prompter and log them to a file.

2.4 Chapter 13. Performance

This chapter discusses the performance issues associated with building VisualAge Generator GUI applications. The chapter covers ways of analyzing the performance of your application and the techniques to use (and not to use) to improve it.

The emphasis of the chapter is on optimization techniques for GUI client applications. Most of the techniques should be incorporated into your everyday coding habits. The exceptions, of course, are those techniques that have side effects, such as decreased readability or significantly increased storage requirements. Such techniques should be reserved for use only after performance bottlenecks have been identified.

Probably the foremost goal of any Generator/4GL product is to increase your productivity. To achieve this goal, you are shielded from seeing much of the *system-level* workings. Examples abound within VisualAge Generator. You do not see the allocations and frees of storage--that is handled by the code that is generated. Likewise, you may not realize how much code must be executed and/or generated to support a single visual connection or a single VisualAge Generator statement.

Thus you might exploit some powerful feature within the product, such as perform request, that, unknown to you, requires that thousands of lines of code be executed to get the job done in the generated code. There may be other, more efficient ways of achieving the same result. If this piece of code is not in a time-critical section of the application, everything is fine. However, if the code is time-critical, the use of this particular technique could result in a serious performance degradation.

Thus, you must balance productivity, reuse, and ease of use against performance requirements--the age-old "software engineering trade-off."

Subtopics

- 2.4.1 GUI Load Time
- 2.4.2 GUI Size
- 2.4.3 Logic
- 2.4.4 Client/Server
- 2.4.5 Generation Options
- 2.4.6 Tuning Runtime Performance
- 2.4.7 Summary
- 2.4.8 What You Should Now Be Able To Do

2.4.1 GUI Load Time

The wait time before a window comes up for an end user is one of the foremost measures of your success in creating an application that performs well. In this section we look at how you can make your windows appear as quickly as possible.

Subtopics

- 2.4.1.1 Loading GUI Applications
- 2.4.1.2 Preloading GUI Applications
- 2.4.1.3 Effect of Parts
- 2.4.1.4 Using User Think Time
- 2.4.1.5 Dynamic Programming

2.4.1.1 Loading GUI Applications

Every time a GUI application is started, a number of parts must be loaded into memory. In fact the first time you start VisualAge Generator runtime services, by issuing an EZE2RUN command, it will load the entire runtime image into memory. This image is about 3MB in size. Any parts loaded thereafter are added on top of this image. These parts make use of the general VisualAge Generator runtime image.

The following rules apply to the timing of the loading of a GUI application into memory. The GUI application is loaded into memory when:

- EZE2RUN opens the GUI application as an application, using the OPEN option. This is the case for the entry point GUI application in your system.
- EZE2RUN explicitly preloads a GUI application into its image, using the LOAD option. EZE2RUN can be issued with this command before the opening of the entry point (even on startup of the machine). Besides specifically using the LOAD option, you can also pass EZE2RUN a file name:

EZE2RUN FILE filename

In the file you can indicate which files to load and which to open. The file will have the following format:

LOAD APP1
LOAD APP2
OPEN ENTRYAPP

- A GUI application is created, for example, using the createPart action, openWidget, new from an object factory, or backgroundCreatePart, and the GUI application was not previously loaded into memory. The GUI application part is created only if it was not previously created or was previously destroyed.
- It is an embeddable part that is contained in a GUI application that is loaded and it was not already loaded into memory.

A GUI application also must be made visible to the user. This opening of the window also costs time. It costs less time when the GUI application is already loaded into memory. For example, if you open a GUI application from another window, close it, and then reopen the GUI application, the time associated with opening the GUI application the second time (not the time associated with loading it into memory--this is obviously zero since it is still in memory) can be as much as 50% less. The same is true if you have preloaded the GUI application, using the LOAD option of EZE2RUN.

You can analyze the loading and opening time by looking at the TSCRIPT.LOG file after you have run an application with the PROFILE and BENCH options of EZE2RUN set ON (see "Runtime Trace" in topic 1.5.3 for a more detailed discussion of the TSCRIPT.LOG file).

2.4.1.2 Preloading GUI Applications

This is a little known secret but you can actually do something very interesting with the VisualAge Generator GUI runtime image. If you are constantly loading the same common parts in your system you can bind them to the GUI runtime image permanently and not have to load them individually anymore. Do this judiciously and keep a backup copy of the original runtime image since that is going to be the only way you can back these saved parts back out of the image.

The way to do this is by passing a SAVE command to EZE2RUN after doing a bunch of LOADs.

You can even put the required LOAD commands and a SAVE command into a file which you run every time you update or install a version of VisualAge Generator.

We recommend that you start from a fresh GUI runtime image when doing this to avoid leaving junk in your image. However, a LOAD command for a GUI that has been presaved into the image should do a destructive load and override the previous copy.

You should also avoid doing a SAVE on a GUI runtime image after doing any OPENs as that would save object instances; they are not something you want in your runtime image.

2.4.1.3 Effect of Parts

The time required for loading and opening a GUI application depends on the parts that are present within the GUI application. Table 12 provides a rough indication of the relative timing of parts that were included on a window. It shows the percentage increase of the load and opening times when a part is added to a window in relation to opening a GUI application with only a window. The figures purely give an indication.

Table 12. Relative Load and Opening Times of Parts		
Part	Load (%)	Open (%)
push button	1	28
toggle button	2	37
radio buttons	4	61
scale	5	46
slider	6	82
hotspot	4	28
entry field	6	30
formatted text	9	42
multi-line edit	3	82
label	5	28
spin button	4	58
list box	5	47
multi-select list	5	47
multi-select list (3 entries)	3	56
drop-down list	2	44
combo box	1	61
container details (1 column)	10	177
container details (2 columns)	5	188
form	3	18
group box	7	25
scrolled window	4	30
PM notebook (1 page)	3	251
PM notebook (2 pages)	5	272
Windows notebook (1 page)	5	114
Windows notebook (2 pages)	12	126
container (1 gadget)	10	107
container (2 gadgets)	16	112
OS/2-Windows notebook (1 page)	12	84
OS/2-Windows notebook (2 pages)	12	84

The time required to load an application is not directly associated with the file size of the .app file that contains the generated Smalltalk code for the GUI application. The .app file can be relatively small but use a lot of features from the basic VisualAge Generator runtime image. That said, a larger .app file will take more time to load most of the time, because there are more parts to create.

Any parts that are placed on the free-form surface do not noticeably affect the time associated with loading the GUI application but parts that are placed on the window do. This distinction is also true for embeddable parts, although these still need to be loaded into memory on their own.

Parts on the free-form surface do have to be created, and this can take quite a bit of time. Remember, you have to load and then create before you can open. Processes, statement groups, and working storage records on the free-form surface can represent a significant portion of the time required to load, create, and open a GUI application. To reduce the influence of logic and data parts, consider these techniques:

- Use a common entry point process or statement group for procedural logic. Only this logic part has to be on the free-form surface, the other logic parts can be executed by name. This removes some logic parts from the free-form surface and can also reduce the visual clutter.
- Reduce the number and complexity of working storage records on the free-form surface. Define smaller working storage records with a specific scope. The process of creating the interface for all data items in a working storage record is very expensive.

Embeddable parts have many benefits, especially from a maintenance perspective. However, there is an additional cost when the application is initially loaded (because each embeddable part is in a separate file), and there can be a performance and memory hit on subsequent runs if the embeddable parts were not well architected (not truly encapsulated).

To balance the benefits and cost of embeddable parts, ensure that their use is warranted and they are well designed. In other words, ensure that the GUI being embedded is or will be used elsewhere and shared data references are implemented with variable parts. If multiple, different embedded GUIs are similar in function, consider generalizing them into a single, general-purpose, embedded GUI. This will provide even greater reuse benefits, while the memory and performance cost, if reused effectively, will be justified.

2.4.1.4 Using User Think Time

Users do not continuously interact with the system. They may be browsing through their papers, looking at the screen absorbing the information, or thinking. You can use this time to your advantage in your application design.

For example, the backgroundCreatePart action allows you to utilize user think time and/or server call wait time to do something useful; namely, the target GUI application of the backgroundCreatePart action will be loaded (if not already) and the Smalltalk parts will be created. Once completed, a subsequent openWidget has far less work to do.

The benefits of using backgroundCreatePart, like preloading GUI applications, can be dramatic. However, use this feature with discretion because memory is required to hold the newly created part. If the application never or only rarely uses a particular GUI application, it is probably not a good candidate for backgroundCreatePart.

2.4.1.5 Dynamic Programming

VisualAge Generator supports dynamic programming in a GUI application environment. Thus you can create additional objects and modify the visual view of these objects during runtime. In this section we discuss two approaches for dynamic programming in VisualAge Generator.

Subtopics

2.4.1.5.1 Object Factory

2.4.1.5.2 Adding Parts to Another Part

2.4.1.5.1 Object Factory

The difference between opening a GUI application directly, using openWidget (when it still needs to be loaded), and using an object factory for this purpose is relatively small. Because using an object factory provides you with more flexibility in your programming, we suggest using the part for opening most of your external GUI applications.

The same considerations you make when creating an embeddable part (whether or not to preload and so on) have to be made when you dynamically add a part, using the object factory.

Using the object factory ensures that the user can open multiple instances of the same window and even allows you to dynamically determine which window to open. You will of course need to add some facility for managing the different windows (for example, their parent-child relationship) and the way in which data is passed between them.

A side benefit of using the object factory is the potential to reduce the number of connections and the complexity of the GUI application. This may result in better performance and easier maintenance.

2.4.1.5.2 Adding Parts to Another Part

It is possible in VisualAge Generator to add parts from the free-form surface to a window. This is like opening a new widget on the window and incurs almost the same performance cost as the opening of the widget would have cost if it had been included on the window from the start. Changing the location of a part dynamically does not incur a very high cost.

2.4.2 GUI Size

The size of the generated .app file is directly associated with the parts that are contained within it. Each part has its own base size. The actual size depends on the length of the part name and the number of features that have been overridden from the default settings.

The size of a part should be factored in to your view of how to write better-performing GUI applications. The .app file is compressed; it expands as it is loaded into memory. If you have replicated data buffers (working storage records) all over your system, the memory size of the runtime environment is going to grow rapidly. Smaller parts mean less memory consumed.

The memory size and the amount of swapping that occurs within your operating system can be controlled to fine tune your system (see "Tuning Runtime Performance" in topic 2.4.6).

2.4.3 Logic

Besides the amount of time it takes to load a GUI application, it is also important to consider the amount of time associated with different actions. Actions can also have an impact on the time it takes to load a GUI application if they occur when the GUI application is opened. In this section we focus on several issues to consider when using visual programming.

Subtopics

- 2.4.3.1 Procedural Logic Entry Points
- 2.4.3.2 Connections between GUI Applications
- 2.4.3.3 Unidirectional Connections
- 2.4.3.4 Eliminate Visual Connections for Control Flow
- 2.4.3.5 Use Special Actions to Populate Table and List Parts
- 2.4.3.6 Perform Request Considerations

2.4.3.1 Procedural Logic Entry Points

Consider reducing the number of logic parts on the free-form surface by using a common entry point. Multiple events that require procedural logic for implementation could trigger the execute action of one process. This process (the common entry point) could then check data item values and perform the required processing. Reducing the number of logic parts on the free-form surface improves part creation time (see "Effect of Parts" in topic 2.4.1.3).

If the one invocation of the common entry point can implement the processing required for multiple logical user events, this can greatly improve GUI application performance. Performance is improved by reducing the amount of time spent synchronizing data in a GUI application. Data is synchronized at the end of each logic part that is triggered with the execute action. If one logic part invokes another logic part there is no synchronization at the end of the invoked set of logic.

2.4.3.2 Connections between GUI Applications

Sharing information between GUI applications is necessary. However, you should minimize the number of other connections to that shared information. When a GUI application opens another, all attribute connections between the applications are triggered. If the shared information also has connections to other information, those connections are triggered as well, basically creating a chain reaction. This occurs when another GUI application is opened, as well as when the data changes.

GUI PROGRAMMING EXAMPLE: USING DATA TRIGGERS AND CONNECTIONS: PART 1

Let's try to explain the cascading of data changes a bit further, using an example:

1. Create two GUI applications (**TGRGUI1** and **TGRGUI2**). Put a push button on the window of each GUI application and place the same working storage record on the free-form surface of each. The working storage record only has to contain one data item. (You can use the record created during Using a Data Item Toggle to Open a Message Window in topic 2.4.3.1, if you wish.)
2. Place an entry field connected to the record data item on the first GUI application (**TGRGUI1**). Disable the **Signal events on each keystroke** setting of the entry field.
3. Add the second GUI application (**TGRGUI2**) as an external GUI application on the free-form surface of both applications.

Note: Embedding an application within itself may give you a warning that you may be creating a recursive call. In this case we control the opening of the window using a push button so we run no risk of creating a recursive call.

4. Connect the clicked event of both push buttons to the openWidget action of the **TGRGUI2** external GUI application.
5. Promote the data attribute from the working storage record of the second GUI application to the first. Connect the data attribute of the working storage record on both GUI applications to the promoted feature from the GUI application.
6. Test the **TGRGUI1** application to see whether it works.

You should be able to create an unlimited number of windows. If you change the data in the entry field in the first GUI application, it should change in all working storage records throughout the system (although you cannot see this unless you use the Test Application data... menu option).

7. To understand the overhead of all of the data synchronization you can use one of the following techniques:
 - a. Turn on test run tracing, open up a Trace Log window, and watch the connections triggered when you modify the data in the entry field and tab out. Try this with a small number of open windows first, then open more and try again.
 - b. Generate the application and run it with the PROFILE and BENCH options set on. Open about 10 windows and then enter a text string in the entry field. Tab out of the field and close all windows.

Once you have run the application and stopped the runtime image, open the Test Application data... menu option and take a look at it.

You will notice that every time you opened a window the data-to-data connection fired on all the GUI applications that were already open. This also happened when you changed the contents of the entry field. You can imagine what kind of activity would occur in your system if you had a lot of these kinds of connections.

The use of synchronized data across multiple GUI applications is one of the most common performance problems. Fortunately, there are other options.

GUI PROGRAMMING EXAMPLE: USING DATA TRIGGERS AND CONNECTIONS: PART 2

Let's look at an alternative that uses variable parts.

1. Replace the working storage record in the **TGRGUI2** GUI application with a variable part. **Build features based on member...** to have the variable part give you the same feature for the working storage record when you look at the Connect window. Promote self attribute to variable part.
2. Remove the previous data-to-data connections between the working storage record and external GUI application, **TGRGUI2**, in both **TGRGUI1** and **TGRGUI2**.
3. Connect the self attribute of the variable part to the promoted feature of the external GUI application in **TGRGUI2**.
4. Connect the self attribute of the working storage record to the promoted feature of the external GUI application feature of the external GUI application in **TGRGUI1**.
5. Use the ITF or generate the GUI applications (as described in Using Data Triggers and Connections: Part 1 in topic 2.4.3.2) and evaluate the changes in runtime trigger.

We can see that the connection between the **TGRGUI1** and **TGRGUI2** GUI applications and the working storage record and **TGRGUI2** GUI applications still all fire each time a GUI application is opened. Even when we update the data in the entry field, this event is only signaled once. This is due to the fact that you are changing the data at a certain memory location and the variable part is nothing more than point at that memory location. The memory location itself did not change, so the connections did not get triggered.

But why do we still need this synchronization of the GUI applications every time they are opened? Let's take our example one step further to look at this behavior.

GUI PROGRAMMING EXAMPLE: USING DATA TRIGGERS AND CONNECTIONS: PART 3

Let's look at an alternative that manages the timing of data synchronization.

1. Delete the self attribute to promoted feature connections in both **TGRGUI1** and **TGRGUI2**.
2. Make a connection between the clicked event of the push button in **TGRGUI1** and the promoted feature of the **TGRGUI2** external GUI application.
3. Provide the self attribute of the working storage record (in **TGRGUI1**) or the variable part (in **TGRGUI2**) as the parameter to this connection.
4. Use the ITF or generate the GUI applications (as described in Using Data Triggers and Connections: Part 1 in topic 2.4.3.2) and evaluate the changes in runtime trigger.

This time, on opening a new GUI application, only one connection is triggered. The connection is that any time you use multiple GUI applications, whether they are embedded or external, should use event-to-attribute connections for the transfer of data between these applications. Also use a variable part whenever possible. This will keep the data across all GUI applications identical without the need for updating each one.

The technique demonstrated in Using Data Triggers and Connections: Part 3 in topic 2.4.3.2 is generally a good solution. But you cannot access a variable part in a process or statement group. In these situations you have to move the data to a working storage record before using it in a process or statement group. This would again cause a lot of connections if you are trying to implement data that is global to the entire application. Is there another option?

GUI PROGRAMMING EXAMPLE: SHARING DATA WITH A VISUALAGE GENERATOR TABLE

Let's look at how we can use a VisualAge Generator table to share data. Remember a VisualAge Generator table can be shared among GUI applications without connections.

1. Save the **TGRGUI1** and **TGRGUI2** GUI applications as **TGRTAB1** and **TGRTAB2**.
2. Change the references to **TGRGUI2** to **TGRTAB2** in both the **TGRTAB1** and **TGRTAB2** GUI applications.
3. Create a VisualAge Generator table (**TGRTBLE**) with the same structure as the working storage record.

record. (If you used the **TRIGGER** working storage record, just add the message data value as the VisualAge Generator table contents. Add a row with some data value as the VisualAge Generator table contents.

4. Replace the variable part and working storage record with the VisualAge Generator table in both the **TGRTAB1** and **TGRTAB2** GUI applications.

5. Quick Form the VisualAge Generator table to the window in both the **TGRTAB1** and **TGRTAB2** GUI applications. Delete the entry field in the the **TGRTAB1** GUI application.

A VisualAge Generator table is always an array, so the Quick Form of the VisualAge Generator table will create a container details. You can use this to change the VisualAge Generator table data at runtime.

6. Use the ITF or generate the GUI applications (as described in Using Data Triggers and Connections: Part 1 in topic 2.4.3.2) and evaluate the changes in runtime triggering. Change the contents of the container details cell and then see whether this change is visible in the other GUI applications you have open (it should not be).

"Touch" the container details cell in one of the other GUI applications and then evaluate. You should now see a change in the container details cell data value.

You will notice that using a VisualAge Generator table causes very little action. There is no need for synchronization when a GUI application is opened and no need to signal the other GUI application that the data of the VisualAge Generator table has changed.

This directly comes back to the difference in how you use VisualAge Generator tables. You do not explicitly need to refresh any attribute-to-attribute connections to the data in the VisualAge Generator table. The VisualAge Generator table does not signal, across GUI applications, that the VisualAge Generator table data has changed. Signals of data changes do occur within a single GUI application.

This ability to control cross-GUI application triggering is one of the subtle features of the VisualAge Generator table that can prove quite valuable if integrated into your GUI application system architecture.

2.4.3.3 Unidirectional Connections

An attribute-to-attribute connection can be made unidirectional, causing only changes in the source to be transferred to the target attribute. To make a connection unidirectional, open the settings for the connection and check the **Read-only source** toggle button.

Use unidirectional attribute-to-attribute connections where appropriate to reduce the amount of signaling.

2.4.3.4 Eliminate Visual Connections for Control Flow

Connections are not always your only option for implementing application function. We discuss alternatives to common connection-based programming techniques in this section.

Subtopics

2.4.3.4.1 Process to Process

2.4.3.4.2 Iterator

2.4.3.4.1 *Process to Process*

Visual programming does some things exceptionally well. Other things, such as control flow, are best done within a procedural language. For example, connecting *has executed* of one process to execute of another process is roughly 10 times slower than coding a PERFORM of the second process within the first one.

There are times when this visual control flow logic is appropriate, especially if other actions are also dependent on has executed of the processes. However, because of the magnitude of the performance cost, use process-to-process connections only when absolutely necessary.

2.4.3.4.2 Iterator

The iterator is a useful part for creating visual loops. Its implementation causes a lot of events to be signaled, however. Do not use the iterator as a substitute for writing a loop in procedural code; reserve it for special circumstances, such as when you want to perform an action other than changing the content on every element in the ordered collection.

See Using the Iterator in topic 1.9.3.1 for a GUI programming example that uses the iterator.

2.4.3.5 Use Special Actions to Populate Table and List Parts

Use the following special actions when populating a Table or List part:

- ☐ getFieldsStartingAt:to:
- ☐ getValuesStartingAt:to:
- ☐ setValuesStartingAt:to:using:

These actions are available from the tear-off attribute of the occurs item (array) in the VisualAge Generator record member and from the table columns attribute of the VisualAge Generator table.

The use of the occurs item actions is much more efficient than connecting directly to the occurs item. For example, the getFieldsStartingAt:to: action copies from 1 to n items of the array. By contrast, if the items attribute of the occurs item is connected to the rows attribute of the Table part, the entire occurs item (array) is scanned, and all items up to and including the last nondefault item are copied to the Table. In an array with an occurs value of 50, if you connect the items to the Table, VisualAge Generator looks at each data item in the array to see if it is a nondefault value before displaying the values. If the getFieldsStartingAt:to: action is used, VisualAge Generator copies the 50 items without the overhead of checking for a default value. Zeroes or blanks will show in this case. If the application can determine the number of valid data items in the array, you can set parameters for the getFieldsStartingAt:to: action, further accelerating the Table population process.

An event needs to trigger the occurs item actions. Often the best event is the has Executed of the Logic Member part in which the data for the Table is accumulated or created. Then define the parameters as appropriate in the Settings or with connections to the parameters on the event-to-action connection. Finally, connect the rows of the Table to the result of the event-to-action connection.

If there is also an occurs item connected to the collection representing the selected rows or items (such as selectedIndices or selectedRows), these special actions should be used for connecting to this occurs item also. Otherwise, the time needed to process the occurs item for the selected rows or items will impact performance.

2.4.3.6 Perform Request Considerations

Perform request processing, while not always measurably significant, does represent overhead if used inappropriately.

Subtopics

2.4.3.6.1 Limit the Number of Occurs in the Perform Request Request Object

2.4.3.6.2 Limit Perform Request Usage

Limit the Number of Occurs in the Perform Request Request Object

2.4.3.6.1 Limit the Number of Occurs in the Perform Request Request Object

When searching an occurs item that is a parameter for a performRequest action, VisualAge Generator searches all occurs values to determine whether an action is to take place. The search does not stop at the first blank action. Therefore, the structure should be as small as possible to avoid the overhead of searching through blank actions.

For example, suppose an application system consists of 10 GUI applications. If 5 of the GUIs require 1 performRequest action, 4 require 2, and one requires up to 10, consider creating three performRequest records. The first will have 1 occur, the second will have 2, and the third will have 10. Additionally, if the GUI requiring up to 10 performRequest actions really has several code paths, some of which only do one or two performRequest actions, using multiple performRequest records within the same GUI can improve performance (although memory usage will increase a bit).

Working storage records used for performRequest processing should be separate and not merged with other general purpose working storage records.

2.4.3.6.2 Limit Perform Request Usage

Although certain operations may seem easier to implement using the performRequest action, its usage should be limited. PerformRequest is a powerful feature because it enables you to create and send a message dynamically. However, this dynamic capability has a fairly high cost. In the ideal situation where all performRequests can be removed, the application will become noticeably smaller and faster.

Try using data triggers (see "Data Item" in topic 1.9.2.2) instead of a performRequest action when possible.

2.4.4 Client/Server

When developing any client/server application system, it is important to consider performance. Of course, one could say the same thing of any software program, whether client, server, or stand-alone. However, a client/server system introduces processing characteristics and possibilities for communicating between machines that are not encountered in a stand-alone environment.

Additionally, client/server topologies likely introduce the use of workstations as opposed to relying solely on midranges and mainframes. Although workstations have increased significantly in processing power since their introduction, they still are far slower than their "big-iron" siblings. As a result, function that was once delivered with acceptable performance in a stand-alone fashion on MVS may not perform acceptably in a client/server environment without explicit attention to design and implementation techniques.

To optimize the performance of any client/server system, these high-level guidelines should be followed:

- ☐ Developers should use good client/server design and coding practices.
- ☐ Human factors engineers should work with developers to create simple, easy-to-use GUIs.
- ☐ System administrators should tune the system and network.
- ☐ Database administrators should tune the database and work with developers on optimal access techniques.

Subtopics

2.4.4.1 Design Considerations

2.4.4.2 Data Considerations

2.4.4.1 Design Considerations

In a client/server environment, computing activities are carried out where they can be performed most efficiently and effectively. Therefore the core issue of a client/server design is deciding where code is going to be executed and the performance characteristics associated with that decision.

Subtopics

2.4.4.1.1 Minimize or Eliminate Server Calls

2.4.4.1.2 Filter or Format Data on the Server (Closer to Source)

2.4.4.1.3 Minimize the Amount of Data Passed between Client and Server

2.4.4.1.1 Minimize or Eliminate Server Calls

Users use a system to perform business transactions. These transactions are the atomic structure of your application design. The GUI application is the front end for the business transaction.

A business transaction consists of business logic (does the customer have enough credit for this transaction?) and accesses data (how much credit does the customer have?). Often a business transaction consists of multiple logical components.

In a client/server environment it is best to execute all of the logical components in one server application. This approach ensures that you cross the network boundary only once during the call to a business transaction. The network is probably one of the slowest components in your architecture.

One of the big advantages of VisualAge Generator is the fact that you can execute the business transaction on the server platform. Imagine the following situation: To enter a new flight into its reservation system, a travel agency has to look up and update about 20 database tables. The flight has to be entered into the reservation system for each day of an entire season (around 200 days). Therefore, if you were not able to write this code to execute on the server platform, you would cross the network boundary 4000 times for this single business transaction. For a single transaction, this is very slow; combined with all other transactions by all other users, it is unrealistic.

In all cases where a business transaction includes more than one server call, combining the calls into a single call, an *umbrella server* application that calls the multiple applications in turn, can result in noticeable improvements.

This corresponds to the 3-tier architecture suggested for client/server applications, which divides presentation from business logic (umbrella server) from data access (atomic server). Figure 53 shows the general structure of a client/server application designed using a three-tier approach.

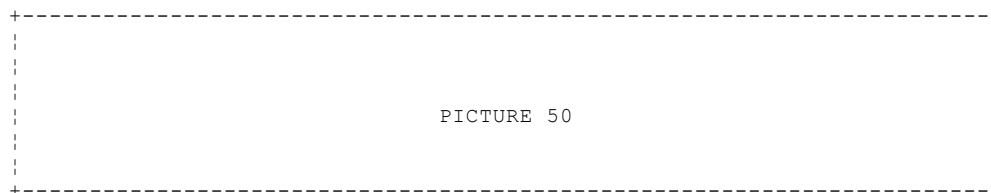


Figure 53. Three-Tier Architecture of Client/Server Applications. A client/server application system can have a three-tier design but not be implemented in a three-tier environment. If both the umbrella and atomic servers run on the same server platform, the implementation is two-tier. One of the strengths of a three-tier design is its ability to use both two-and three-tier implementation configurations.

2.4.4.1.2 *Filter or Format Data on the Server (Closer to Source)*

This technique exploits the higher processing power of the server platform, assuming the server **is** more powerful than the client. With filtering (reducing the answer set), less data has to be passed between the server and client machines. There really are no drawbacks because filtering saves bringing down unnecessary information.

With formatting (changing the answer set), multiple client applications may be simplified if they all need the same form of the data. One can argue whether formatting should be done closer to the source (server database) or closer to the target (GUI). As is often the case, there is no single "right" answer. If formatting adds significantly to the size of the answer set, the additional communications overhead might suggest that it remain on the client.

2.4.4.1.3 Minimize the Amount of Data Passed between Client and Server

It is important to minimize the amount of data passed across the network to reduce network traffic. Also, if data conversion is done, each byte is converted going from client to server and back, even if it is a blank. Whenever the amount of data can be reduced, the client transaction time will improve.

Particular attention should be given to the size of arrays within the answer set. If some server calls return relatively few rows of data and others return many rows, consider using separate records to handle each case. Additional record definitions will affect maintenance and memory requirements.

2.4.4.2 Data Considerations

The approach you use for data management and the design of your GUI applications, working storage records, and server interactions have a significant influence on the overall performance of your system. This section reviews some approaches and techniques for improving data management and system performance.

Subtopics

2.4.4.2.1 Cache, Cache, Cache

2.4.4.2.2 Array Dimensions

2.4.4.2.3 Level-77 Data Items

2.4.4.2.4 Substructured Records and Occurs Items

2.4.4.2.5 Data Types

2.4.4.2.1 Cache, Cache, Cache

Caching is the fastest and easiest way to improve performance in systems that are not memory-constrained. Savings can be quite significant because subsequent retrievals or recalculations of data are eliminated. However, caching has a potentially large drawback because it can add significantly to the complexity of an application. Specifically, if the cached information changes frequently, there is the possibility that the cache will get out of sync with the original source. Thus, the rate of change of the source is inversely related to the applicability of caching it elsewhere.

Cache can be implemented in a number of ways, but obviously the ordered collection (because of its unlimited storage space and independence of its contents) and the VisualAge Generator table (because of the fact that its contents are automatically shared between GUI applications) are potentially the most powerful parts in VisualAge Generator for usage as cache.

Let's look at two examples of using cache. In the first example, we look at a possible elimination of backward scrolling in cases of megadata, and in the second example we look at a way of implementing lookup tables at runtime.

The first example is Dealing with Megadata in topic 2.4.4.2.1.

Megadata concerns those situations in which you fetch data from a data source but do not know exactly how much data will be returned, or the amount of data that is returned is larger than the allowable or preferable size. Typically this issue is resolved by implementing a paging mechanism. The user can fetch the data in pages (whether this is evident to the user or not in the implementation of the user interface is not relevant for this discussion).

Each page contains a fixed number of rows. Getting the next page involves your knowing the location of the last row in the previous page. This row can be uniquely identified by the values for the data items of the primary key of that row and the values for the data items in the sort key for that row. Fetching a next page is then searching for all values greater than or less than (depending on the sort order) this key.

Getting the previous page involves exactly the opposite mechanism against the first row in the previous page. But getting a previous row seems strange because you have already fetched the row, so why should you fetch it again? Especially if you are not memory constrained on the client for holding large amounts of data. Although you will not get the latest information from the database when scrolling back using the previous data (this facility is reserved for a refresh option of the list), you will be able to respond quickly to a user request.

```
+-----+
| GUI PROGRAMMING EXAMPLE:  DEALING WITH MEGADATA
```

This example provides an understanding of how you can deal with Megadata. We will not use a server but we will use a process to either fill a working storage record with the next data, if paging forward, or the previous data in the working storage record, if paging back.

Note: A very similar example is provided in the VisualAge Generator Developer sample as **PAGING.ESF**.

1. Create a new GUI application. Add a list box to the window and two push buttons, labeled "Previous" and one labeled "Next."
2. Drop a working storage record, **MEGADATAWS** with the following structure on the fre surface:

10 ROW	char 3	occurs 20
12 NUMBER	num 3	
10 PAGE	num 3	
10 PAGES	num 3	
10 INDEX	num 2	
77 TRIGGER	char 1	
77 TRIGGER2	char 1	
3. Connect the ROW data attribute of the working storage record to the items attribute list box.
4. Create a process named **MEGA-PROC-LOAD** with the following statements:

```
IF PAGE GT PAGES;
  TRIGGER2 = "Y";
END;
```

5. Create a process named **MEGA-PROC** with the following statements:

```
PAGE = PAGE + 1;
IF PAGE GT PAGES;
  INDEX = 1;
  /* Imagine that this is your call to a server
  /* application
  WHILE INDEX LE 20;
    NUMBER(INDEX) = (20 * PAGE) + INDEX;
    INDEX = INDEX + 1;
  END;
ELSE;
  TRIGGER = "Y";
END;
```

.

.

6. Connect the aboutToOpenWidget event of the window to the execute action of the **MEGA-PROC-LOAD** process.

7. Connect the clicked event of the **Next** push button to the execute action of the **MEGA-PROC-LOAD** and **MEGA-PROC** processes, in that order.

8. Create a process named **MEGA-PROC-BACK** with the following statements:

```
IF PAGE GT 1;
  PAGE = PAGE - 1;
  TRIGGER = "Y";
END;
```

9. Connect the clicked event of the **Previous** push button to the execute action of the **MEGA-PROC-LOAD** and **MEGA-PROC-BACK** processes, in that order.

10. Put an ordered collection on the free-form surface.

11. Connect the size attribute of the ordered collection to the enabled attribute of the **Previous** push button. This will disable the push button when there are no entries in the ordered collection (size is zero and zero is false). Once the size grows the push button is enabled (any value greater than zero is true). We do not disable the push button on the example if you scroll forward and then backward.

12. Connect the size attribute of the ordered collection to the PAGES data attribute of the working storage record.

13. Connect the TRIGGER data event to the atIndex: action of the ordered collection. Connect the PAGE data as the parameter. Connect the result attribute of the connection to the result attribute of the working storage record.

14. Connect the TRIGGER2 data event to the add: action of the ordered collection. Connect the ROW data attribute of the working storage record as the parameter.

15. Test the application (save the GUI application as **MEGADATA**).

In this example we do not really go to a server application but if we would we would have an enormous amount of calls to server applications that do nothing else than fetch data that we had already fetched.

HINT

Do not store the data attribute of a working storage record in an ordered collection. You will not be able to access it. Always use a substructured data item for this purpose.

Most applications use some form of lookup data, for example, airport codes, descriptions for codes, message codes. This data does not change as often as other data, and you could presume that it stays the same during the duration of a session when the user uses the application. The data may also be used in several places throughout the application.

In these situations it could be useful to fetch the data only once, when it is accessed for the first time. This can be done in VisualAge Generator with VisualAge Generator tables. (13)

GUI PROGRAMMING EXAMPLE: IMPLEMENTING LOOKUP TABLES AT RUNTIME

This example shows an implementation of shared lookup data support. We use a VisualAge Generator table because it is very efficient at sharing data in a runtime environment.

1. Create a new GUI application. It will be used as a non-visible part. Delete the default label and add a label as its primary part. Provide the label with an appropriate name such as "AirportData."

2. Create a VisualAge Generator table named **AIRPORT** (originally created in Validating VisualAge Generator Tables in topic 2.3.2.2.3) with the following structure:

```
10 AIRPORTCODE      char 3
10 AIRPORTNAME      char 50
```

3. Add 20 blank rows to the VisualAge Generator table. The VisualAge Generator table has a static size so this must be defined before you start out using it.

4. Create a process with the following statements (this process simulates a server call):

```
AIRPORTCODE(1) = "SFO";
AIRPORTNAME(1) = "San Francisco International";
AIRPORTCODE(2) = "SJC";
AIRPORTNAME(2) = "San Jose (CA)";
AIRPORTCODE(3) = "AMS";
AIRPORTNAME(3) = "Amsterdam Schiphol";
AIRPORTCODE(4) = "DFW";
AIRPORTNAME(4) = "Dallas/Fort Worth";
AIRPORTCODE(5) = "RDU";
AIRPORTNAME(5) = "Raleigh/Durham";
```

5. Add a toggle button to the free-form surface of the GUI application. Promote the button as getAirportData. Promote the enable action as refreshAirportData.

You could just trigger off of the aboutToOpenWidget event of the primary part of the application to initialize the data, but if you were actually calling a server application you may need to synchronize the data every few hours. The distinct event, refreshAirportData provides that flexibility.

.

.

.

.

.

.

6. Connect the selection and the enabled events of the toggle button to the execute event of the process. The selection event only gets signaled when the selection is changed only the first time, since thereafter the toggle button will already have been selected. The enabled event can be signaled multiple times.

7. Save the GUI application as **AIRDATA**.

8. Create a second GUI application and add the embedded GUI application **AIRDATA** and the VisualAge Generator table to the free-form surface.

9. Put three push buttons on the window. Give them labels of Get Data, Load Data, and Save Data.

10. Connect the clicked events of the push buttons to these **AIRDATA** embedded GUI application actions:

Get Data to getAirportData.
Refresh Data to refreshAirportData

11. Quick Form the **AIRPORT** VisualAge Generator table on the window to create a container details. Delete the connection between the VisualAge Generator table and the container details.
12. Connect the clicked event of the Load Data push button to the container details id attribute. Use the table columns attribute of the **AIRPORT** VisualAge Generator table as the parameter.
13. Test the application (save the member as **AIRTEST**).

The data is fetched only the first time the Get Data push button is clicked. This causes the container details to load the data. You can force the load by clicking the Load Data push button. The Refresh Data push button will run the procedural logic. If the data had been changed, you would need to use the Load Data push button again to see the changes.

There are two other ways to see the container details contents after you have clicked the Get Data push button without using the Load Data push button:

- Minimize and then restore the window with the container details
- Cover and then uncover the container details area with another window

This should reinforce the view that you must control the refresh of data from the VisualAge Generator tables that are being used as global shared data without connections.

You could design a VisualAge Generator table as a common source for multiple forms of lookup data. The initialization and retrieval activity could be performed once at system startup. You could then use the shared global data behavior for the VisualAge Generator table part to see this data wherever required in your system without constantly using embedded GUI applications that must be synchronized.

- (13) You could also implement this function with working storage records or parts, although you would have to pass these around in your application by using event-to-attribute connections.

2.4.4.2.2 Array Dimensions

The dimension or size of arrays can have a significant impact on application performance. Thus, attempt to limit the size of arrays such that most cases can be handled. If necessary, a special-case array can be used when the smaller, general-purpose array is not sufficient. If the amount of data that must be processed varies significantly, using a large array to handle the upper bounds is the easiest solution. Multiple record definitions must be maintained if more than one record is used (general and special case).

2.4.4.2.3 Level-77 Data Items

Because level-77 data items are not part of the record structure, their use in GUI logic can be optimized substantially. Specifically, because there is no concern about other items substructuring a level-77 item, VisualAge Generator can cache its value. Therefore, operations requiring use of this value need not perform format conversions (which must occur for substructured regions within records).

Level-77 data items are excellent candidates for counters, temporary variables, trigger data items.

2.4.4.2.4 Substructured Records and Occurs Items

VisualAge Generator can generate optimized code for the singly occurring data items within the *flat* portion of a record. This optimized code allows for value caching, just as is done unconditionally for level-77 items.

Records with substructured items require more processing than flat records of the same size. Thus processing the record to pass on a client/server call can be quite costly. However, additional record definitions will affect maintenance and memory requirements.

2.4.4.2.5 Data Types

VisualAge Generator's GUI runtime optimizes data items with no decimal positions.

VisualAge Generator supports automatic type conversion when moving or comparing data items of different types. However, this automatic conversion is costly and should be avoided when possible.

VisualAge Generator supports automatic truncation and padding when moving or comparing nonnumeric data items of different lengths. However, this automatic truncation and/or padding is costly and should be avoided when possible.

2.4.5 Generation Options

VisualAge Generator generation options influence the generated code and through this the overall performance of the system.

Subtopics

2.4.5.1 Use the NONUMOVFL Generation Option Whenever Possible

2.4.5.1 Use the NONUMOVFL Generation Option Whenever Possible

Performance of GUI applications is significantly faster when the generated client logic does not have to check for numeric overflow. This is especially important in applications containing a significant amount of client logic. When NONUMOVFL is used, a numeric overflow condition will trigger a walkback error message.

2.4.6 Tuning Runtime Performance

This topic was covered by Jon Shavor, a member of the VisualAge Generator development team, in the article "Running Your VisualAge Generator GUI in a Small Footprint" which was included in the *IBM VisualAge Generator Newsletter*, Volume 1, Number 3.

We include it as-is in our book for completeness.

Many of you have a need to run GUI applications on machines with minimal memory. This article provides a few tips to help you accomplish this task, especially if you have a small number of GUI applications that do not manipulate significant amounts of data. Because each GUI application has unique memory requirements, it is not possible to provide absolute numbers.

An IBM Smalltalk application provides runtime support for VisualAge Generator GUI applications. Generated GUI applications are also Smalltalk code that are loaded into the runtime support image. IBM Smalltalk has a dynamic memory management model that strategically allocates and frees memory when necessary. The following terminology will help you better understand the IBM Smalltalk memory model:

Old Space (MO)

Old space is memory space where IBM Smalltalk stores objects that are still used by the system and have been in storage for some time (tenured objects).

The default value for old space is approximately 2 MB larger than the size of the shipped image (EZE2RUN). For VisualAge Generator Version 2.2, this value is about 5.2 MB for OS/2 and 5.0 MB for Windows. The 2 MB are reserved to load and run your GUIs until more memory is necessary.

By default, 6.2 MB (6 MB for Windows) is being reserved for the VisualAge Generator runtime support for OS/2. This does not include the operating system, communications software, and any other applications you are currently running.

New Space (MN)

New space is the memory space where IBM Smalltalk allocates newly created objects. Two memory regions of the same size are allocated to free up memory faster (see scavenging). Objects are moved from a new space to the old space. The conditions in which the movement occurs are when objects have been in storage awhile and when the new space needs more room to allocate new objects. Once specified, the new space size cannot be changed.

The default value for new space is 512 KB. To determine how much actual space is used, you need to double this number. Thus, 1 MB is being reserved.

Memory Increment (MI)

Memory increment is the amount of memory IBM Smalltalk requests from the operation system when more memory is needed. The memory is requested after a garbage collection. If the amount of free space in the old space is less than a certain amount, more memory is requested. This space is added to the old space.

The default value for memory increment is 2 MB.

Scavenge

A scavenge is to copy the objects still used by the system from the currently used new space to the other new space region. This occurs when the amount of free memory available in the new space reaches a specific amount. Thus, any unused objects are left in place, which is similar to freeing the memory. A scavenge is a quick process that can occur frequently. Scavenging is so quick, it is not noticed by the end user. Increasing or decreasing the size of new space results in an increase or decrease in the number of scavenges.

Garbage Collection

A garbage collection is when tenured objects in the new space are moved to old space and all unused objects in the old space are freed. When the availability of free memory in the new space reaches a certain amount after a scavenge, IBM Smalltalk does a garbage collection. If the amount of free memory in the old space reaches a certain amount after a garbage collection, IBM Smalltalk asks the operating system for more old space memory. The memory increment value determines how much memory

is requested.

Compared to scavenging, a garbage collection is rather lengthy. Depending on the size of your image, the numbers will change, but scavenging can be approximately 10,000+ times faster than a garbage collection. A scavenge might take less than a millisecond, whereas a garbage collection can be in the range of 5 to 10 seconds.

VisualAge Generator V2.2 enables you to modify the old space (MO), the new space (MN), and the memory increment (MI). Thus, if you have a relatively small suite of GUI applications and data, the defaults can be reduced to run in a smaller footprint. Alternatively, if you have a very large suite of GUI applications and data, increasing these values can result in dramatic performance improvements.

The following environment variables are supported to manipulate the above values:

EZERRUN_MO

This environment variable enables you to tune the memory requirements for GUI runtime for either the OS/2 or Windows environment. It represents the amount of preallocated memory reserved for tenured objects during GUI runtime. The value is in megabytes (MB).

The default is approximately 2 MB more than the size of the image file (EZE2RUN). The minimum recommended setting is 500 KB more than the size of the image file (EZE2RUN). The maximum recommended setting depends on the amount of memory installed on your machine and the size of the GUIs you are running. If you change this environment variable, you must stop and start the GUI runtime support for the change to take effect.

The following is an example of how to set the MO option to 8 MB:

```
EZE2RUN STOP
SET EZERRUN_MO=8
EZE2RUN START
```

EZERRUN_MN

This environment variable enables you to tune the memory requirements for GUI runtime for either the OS/2 or Windows environment. It represents the amount of preallocated memory reserved for newly created objects during GUI runtime. To calculate the actual amount of memory reserved, multiply this number by 2. IBM Smalltalk reserves two memory areas each of this size for the purposes of scavenging. The value is in kilobytes (KB).

The default is 512 KB. The minimum recommended setting is 256 KB. The maximum recommended setting depends on the amount of memory installed on your machine and the size of the GUIs you are running. Generally, the new space will not need to be larger than 1 MB (1024 KB). If you change this environment variable, you must stop and start the GUI runtime support for the change to take effect.

The following is an example of how to set the MN option to 256 KB:

```
EZE2RUN STOP
SET EZERRUN_MN=256
EZE2RUN START
```

This reserves two 256 KB memory regions for a total of 512 KB of memory.

EZERRUN_MI

This environment variable enables you to tune the memory requirements for GUI runtime for either the OS/2 or Windows environment. It represents the amount of memory to request once the reserved memory is used. The value is in kilobytes (KB).

The default is 2000 KB. The minimum recommended setting depends on the size of the GUIs you are running and the amount of data the GUIs use. 256 KB can be used if your memory requirements are small. The maximum recommended setting depends on the amount of memory installed on your machine and your memory requirements. Generally, the memory increment will

not need to be larger than 2000 KB. If you change this environment variable, you must stop and start the GUI runtime support for the change to take effect.

The following is an example of how to set the MI option to 500 KB:

```
EZE2RUN STOP
SET EZERRUN_MI=500
EZE2RUN START
```

If you have a suite of 5 GUIs that are reasonably small and the GUIs do not manipulate huge amounts of data and you are running the suite on an 8 MB Windows machine or a 12 MB OS/2 machine, to determine an optimal setting, do some trial and error testing. It is recommended that you start by doing the following:

```
SET EZERRUN_MO=3.5 (3.7 for OS/2)
SET EZERRUN_MN=256
SET EZERRUN_MI=500
```

The memory requirements are $3.5 + 2 * .256 = 4.0$ MB. When more memory is needed, 500 KB is requested. If you think you are requesting more memory when your GUIs are initially loaded (you will see extra disk activity), you might need to increase the MO option. It is suggested that you increase the MO option in increments of 500 KB until you think you have found an optimal setting.

Most applications will probably need the MO option to be set to at least 4.0 MB for both Windows and OS/2. It is harder to determine an optimal setting for the MN option. It is recommended that you start with 256 KB for a low memory usage suite. If you find you are starting to use more memory, it is probably more efficient to increase the MN option. Generally, you should need no more than 1 MB of memory. Be careful with the MI option. If you leave MI set at the default of 2 MB, and your suite of GUIs reach the maximum amount, but only by a few KBs, you could be reserving an extra 1.5 MB or more, which you do not need.

For example, you are currently using 3.9 MB of 4.0 MB. If you need an additional 200 KB, which puts you over the amount reserved, you will be using 4.1 MB of 6.0 MB of memory. This is actually OK if you know that your GUI suite will eventually need all this memory. But if you know your GUI suite will not need that much memory, it is wasted memory. It would be more efficient to set MI to 500 KB. Then you would be using 4.1 MB of 4.5 MB of memory. On the other hand, it is more efficient to allocate one larger chunk of memory than multiple smaller chunks. However, make sure you really need the memory you are requesting.

There are other factors that can determine the memory requirements for a suite of GUI applications: for example, proper design of your GUIs and the use of expensive controls. This article does not address these factors. Some of these factors have already been addressed in previous articles and others will be addressed in the future.

In summary, it is possible to run a GUI suite with relatively moderate memory requirements in a small footprint. However, the application developer needs to understand the memory requirements and tune the system accordingly. Hopefully, this article will help you with the tuning of your systems and enable your GUI applications to run on memory-constrained machines.

2.4.7 Summary

Performance considerations are an integral part of application development. When using a product that generates code for you, it is worthwhile knowing which implementations generate better performing code than others. In VisualAge Generator the performance considerations can be broken down into loading a GUI application, GUI application size, logic, and client/server.

VisualAge Generator GUI applications can be preloaded so that when a user accesses a part, it does not have to be loaded into memory again. You can also use the user think time to perform these types of actions.

When implementing logic using visual programming techniques, consider the number of events that the implementation causes and whether they can be reduced. For example, it is better to use an event-to-attribute connection than an attribute-to-attribute connection because you control when it triggers. This is especially relevant in communication between GUI applications. You can also use unidirectional connections if you are sure that the source is read-only.

Visual programming is not always a replacement for procedural coding. Actions that are self-contained and are easily programmed using procedural code should be implemented using processes and statement groups.

Client/server considerations are especially relevant to VisualAge Generator. As the network is one of the slower components in the infrastructure, the number of server calls should be minimized. Combine the business logic and calls to data related applications per business transaction into an umbrella server. Also consider the amount of data that is passed between the client and the server.

Consider using cache to store data that you might need again later. Apparent possibilities for caching include backward paging in a megadata solution and fetching lookup information with a low frequency of change.

Once you have generated your application with the generation options that optimize performance, you can tune the runtime performance of your system by changing the amounts of memory allocated by VisualAge Generator for certain activities. This is a machine-related activity. The settings can be different for different machines in your organization.

2.4.8 What You Should Now Be Able To Do

You should now be able to:

- ☐ Use a number of techniques to ensure that the performance of your application is good.
- ☐ Understand which techniques to use and which not to use.
- ☐ Create a technical design for your application, taking into account performance issues.

3.0 Part 3. Fly

This part of the book covers an architecture for developing VisualAge Generator GUI applications. It explains both the rules that dictate the architecture and the theory on which it is based. The theory will give you further insight into the potential of VisualAge Generator.

This part will enable you to develop VisualAge Generator GUI applications in large-scale application development efforts, providing you with the rules and skills required to build maintainable and reusable code.

Subtopics

- 3.1 Chapter 14. Application Architecture
- 3.2 Chapter 15. Objects
- 3.3 Chapter 16. Objects in VisualAge Generator

3.1 Chapter 14. Application Architecture

Application development requires standards; understandable, maintainable, and reusable code; and discipline. In traditional environments the first subject to be taught has always been structured programming. The goals of the application architecture described herein are to provide you with consistent, maintainable, reusable, and documented applications.

This chapter shows you how to apply structured programming techniques to VisualAge Generator GUI application development. It builds extensively on the theory and practice presented in other chapters of the book, so ensure that you understand the concepts described in those chapters.

First, we describe a general subdivision of applications and the reasoning behind the subdivision. Using the subdivision, we look at the application components and zoom in on their specific requirements and possibilities. Finally, we discuss the interaction of the components and review some issues that are relevant during application development using the application architecture.

The description of each part is summarized as a set of rules to which the resulting application should comply. These rules provide you with a checklist for performing reviews of applications. (14)

(14) In this chapter we limit ourselves to looking at the support of business processes. Building technical embeddable parts to support the applications is expanded on in Chapter 15, "Objects" in topic 3.2 and Chapter 16, "Objects in VisualAge Generator" in topic 3.3.

Subtopics

- 3.1.1 Building from Parts
- 3.1.2 Business Object
- 3.1.3 Representation
- 3.1.4 Controls
- 3.1.5 Common
- 3.1.6 Interaction
- 3.1.7 Application Development Considerations
- 3.1.8 Summary
- 3.1.9 What You Should Now Be Able to Do

3.1.1 Building from Parts

The basic idea behind the architecture is that by building up applications from embeddable parts the complexity of your applications is hidden inside the embeddable parts. By always building up an application from the same types of embeddable parts it is always clear what each part is supposed to do within its context. The part has a kind of contract with the other parts that are used within the system.

The ultimate goal of every business application is to support a business process. The most important part in any application is therefore the part that contains the business logic. Within the application architecture, all the business logic is separated out and is contained within one embeddable part. We call this part the *business object*.

A user of an application system will look at the business in a number of ways. For example, a salesperson at a car dealership may want to show a picture of the car to the customer but may also want to look at specific performance characteristics of the car. Although all of these aspects are related to the same business entity, the car, the salesperson has two different views of it. We call these views *representations*.

A business entity, like a car, is a subject for actions to be performed on it (for example, the car can be sold or brought in for a check up).

Within an application all these actions are performed through either a menu or push buttons. The CUA guidelines even state that any action that can be performed using a push button on a window that also has a menu should also be available in the menu. We call the part with which the user controls the business object the *controls* part.

Finally applications interact with each other. These interactions are characterized by the passing of data and the initialization of other windows. This activity has been categorized as being *common* to the business entity.

Combining these parts forms the basis of each application. Figure 54 shows the general structure of a VisualAge Generator application using the application architecture.

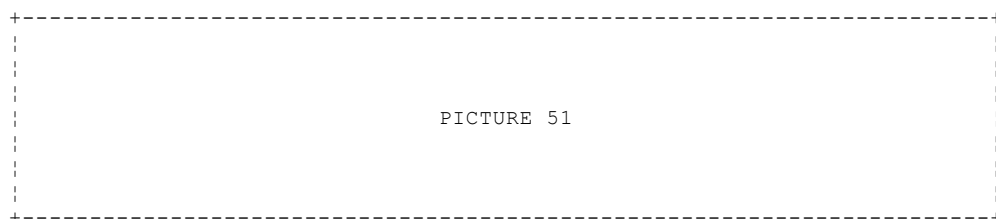


Figure 54. Application Architecture Overview

There are many architectural options, such as that shown in Figure 54, that provide value by standardizing the approach used to design and implement reusable parts for an application system.

There is no one correct approach. Any architecture that focuses on consistent implementation, well-defined boundaries, and reusable parts can be used to implement a well-designed VisualAge Generator-based application system.

3.1.2 Business Object

A business application is used to support a business process. A business process consists of a number of business entities upon which business transactions can be performed. The support of the business process is essentially independent of the way that the user of your application will view the process.

The business object in the application architecture contains the business entities and business transactions. These may reside on the workstation or be represented by implementations of server applications and databases on a host system. One of the good aspects about VisualAge Generator is of course that you are free to choose the platform on which this code resides. Make sure, however, that for performance reasons each business transaction has its own umbrella server.

Subtopics

3.1.2.1 Building the Business Object Part

3.1.2.2 Business Object Part Checklist

3.1.2.1 Building the Business Object Part

The business object by itself does not have a visible component. It is made visible by a using a representation. Therefore we are free to determine the primary part of the business object. We have chosen that the primary part of a business object is a label indicating the name of the business object followed by the name of the VisualAge Generator member. The part name of the label is lbObjectName. Figure 55 shows an empty business object.

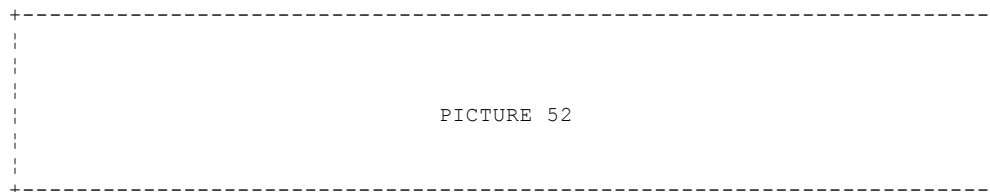


Figure 55. An Empty Business Object

The name of the VisualAge Generator member should comply with the VisualAge Generator naming standards. Appendix B, "VisualAge Generator Naming Convention" in topic B.0 suggests a possible naming convention based on the application architecture.

As business transactions mainly concern accessing data on a server, the business object can be seen as the gateway between your GUI application and the server domain.

The business object contains everything that logically correlates to the business function that is to be performed. The aspects of the business object that are provided should give an interface that describes the business object and neither technical implementation surrounding it nor features that are not essential to the business.

For example, a getCustomerData action would be a valid part of the interface of the business object **Customer**. A feature named ACCSSPG-GET-DATAexecute would violate the restriction that a name cannot be dependent on the technical implementation. A feature named checkCredit is not valid if it does not correspond to a business transaction, as such it might be better placed as part of a method named addCustomer. The action can be part of a business transaction, but then it should not be part of the interface of the business object.

The business object part should either correspond to a single instance of a business object (for example, customer with name "Jansen") or be a container of business objects (for example, a list of all customers). These two business object parts are separate because they have their own set of behavior. For example, you can sort a list but you cannot sort a single instance.

All calls to server applications within the business object must be performed from within a process or a statement group, as opposed to direct visual server calls. You are free in this choice as long as the choice is consistently applied within your organization. Using a process or a statement group allows you to check for errors after the call to the server application returns, by checking either the return codes you pass in a working storage record or the EZERT8 variable. (15)

The business object is a stand-alone entity. It contains the data that is actually associated with it. Therefore it cannot have variable parts. Variable parts are only allowed when a business object contains other business objects to whose data it needs to refer. In these cases the business object contains the actual data in the business object that is embedded in it although it needs a variable part to reference it.

For example, suppose you have an insurance business object that contains a working storage record with all of the base data on insurance. If you create a fire insurance business object, you will want to use the characteristics of the insurance business object and add on to those. Although fire insurance will have its own working storage record with data specific to fire insurance, for any generic insurance information it will refer to the insurance business object. To refer to this information, the fire insurance business object includes a variable part that is connected to the promoted self attribute of the working storage record in the insurance business object.

All working storage records in a business object that are to be part of its interface should have only their self attribute promoted, except when the data is to be passed to other business object parts with which it has a relationship other than being contained in it, as with the insurance and the fire insurance business objects. Because the business object uses a working storage record to physically implement the associated data, when you connect this data with the data implemented in other business objects,

you have to pass the data attribute. You cannot dynamically change the memory location of a part (the working storage record in the two different business objects).

The data does not get passed to other associated parts like the representation part. These parts use variable parts to connect to the promoted self attributes of the data to use the data in the business object.

A business object can contain other business objects. The umbrella servers of these embedded business objects should be combined if an action would cause umbrella servers in more than one of the embedded business objects to be called. This is dictated by the client/server design considerations of your system.

Stated differently: any time an action causes a call to two separate umbrella servers, there should probably be a business object containing the business objects involved.

(15) You must have called the application with the (REPLY option.

3.1.2.2 Business Object Part Checklist

The business object should comply with the following characteristics:

- ☐ There is only one GUI application that defines this business object (this GUI application may be decomposed into other GUI application,s but only the one business object GUI application is embedded when used).
- ☐ The parts in the business object and its interface comply with the naming convention.
- ☐ A container of objects is a separate business object.
- ☐ All server applications should be contained in a business object. A business object can contain multiple server applications.
- ☐ All server applications must be called from either a process or a statement group so that appropriate error checking can be implemented as part of the **CALL** logic. The choice of a process or a statement group should be consistent for your entire organization.
- ☐ There are no variable parts in the business object except if the business object contains other business objects to whose data it needs to refer.
- ☐ The data attribute of a working storage record in the business object is only promoted if the data is to be passed to other business objects. Otherwise self is used.
- ☐ One user action cannot cause more than one call over each platform boundary. In these cases umbrella servers should be combined.

GUI PROGRAMMING EXAMPLE: CREATING A BUSINESS OBJECT

In this example we create a business object for a baby in a nursery. The server appl database are simulated by a VisualAge Generator table.

1. Create a new GUI application.
2. Replace the window with a label. Change the label to "Baby - RBBBZDB."
3. Create a VisualAge Generator table, **RBBBZT1**, with the following structure:

10	BABYID	char	5
10	NAME	char	35
10	BIRTHDATE	char	10
10	WEIGHT	num	4,2
10	MOTHERID	char	5

4. Add a number of entries to the VisualAge Generator table.

One entry should have a BABYID value of '11111' as in:

BABYID	NAME	DATE	WEIGHT	MOTHERID
11111	Brendan Patrick McCarthy	11/14/1996	7.2	22222

5. Create a working storage record, **RBBBZWD**, with the same structure as the VisualAge table.
6. Create a working storage record, **RBBB0WK**, with only the following structure:

BABYID
7. Make sure the VisualAge Generator table and both working storage records are on the free-form surface of the business object. Promote self of the **RBBBZWD** working storage record as babyInformation. Promote self of the **RBBB0WK** working storage record as babyKeyData.
8. Put a process on the free-form surface: **RBBBZP-GETBABY**. Give the process the following code:

```
IF RBBB0WK.BABYID = ' '; /* assign default  
RBBB0WK.BABYID = '11111';  
END;
```


VisualAge Generator GUI Development Guide
Business Object Part Checklist

RETR RBBB0WK.BABYID RBBB0T1.BABYID RBBB0WK.BABYID BABYID;

RBBB0WD.NAME = RBBB0T1.NAME(EZETST);
RBBB0WD.BIRTHDATE = RBBB0T1.BIRTHDATE(EZETST);
RBBB0WD.WEIGHT = RBBB0T1.WEIGHT(EZETST);
RBBB0WD.MOTHERID = RBBB0T1.MOTHERID(EZETST);

9. Promote the execute action of the process as getBaby.

10. Save the GUI application as **RBBBZDB**.

3.1.3 Representation

A user does not interact with a business object but with a representation of the business object. A business object can have several, different, representations. For example, a real estate firm with a database of its real estate may look at a property as a picture, detail information concerning the state of the house, detail information showing the property's price tag, or even a movie that gives a potential house buyer a guided tour of a house.

Each of these representations looks at and should interact with the same business object. If a new representation requires new server logic, you should change the original business object.

Subtopics

3.1.3.1 Building the Representation Part

3.1.3.2 Representation Part Checklist

3.1.3.1 Building the Representation Part

Representations come in various types, so the specific type is not relevant to the architecture. What is relevant is that the representation is used to look at the attributes of the business object. In most cases these are stored as data items in a working storage record.

Therefore the representation part contains parts that point to the attributes that allow you to hook up to the attributes of the business object. If the attributes of the business object have been exposed as the self attribute of a working storage record, you use a variable part in your representation. If they have been exposed in any other way, you provide them as such.

The representation part only contains the controls for viewing the data not, for taking user actions, like opening up a detail window from a list. Menus and push buttons are contained in a separate part. Any user actions that can be taken, such as double-clicking an entry in a list, are promoted as events and should cause the corresponding action in the controls part to be executed.

There are a number of reasons why the subdivision is made in this way. First of all the CUA guidelines prescribe that you can always reach the effect of a direct-manipulation action, such as double-clicking, through the use of a push button and/or a menu. The second reason is that you can have multiple views for each business object. But the actions you can do on a business object are independent of its representation. So whether using a picture view or a detail information view to look at a house, the user should be able to sell the house to the customer.

Since representations contain the data as it is finally viewed by the user, it is also the place where any formatting and editing occurs. Implementation of validation (see "Data Validation Errors" in topic 2.3.2) and the use of reference tables (see "Cache, Cache, Cache" in topic 2.4.4.2.1) are therefore both part of the representation. You can argue about the logical correctness of this choice, but it makes the implementation of your application much more straightforward, hides complexity, and enhances performance.

The interface of the representation part contains features that allow you to pass in the information, refresh the view of the information, and determine the state of the view (for example, the row that is selected in a list). It also contains the menu attributes of those parts that could have an object menu.

Because the representation is affected by the security level of the user, any security mechanism should also be implemented in the representation part (see "Security" in topic 3.3.3.3.2 for a more detailed discussion about the possibilities for implementing a security mechanism in your GUI applications).

If the representation contains multiple parts, these can be combined on a form. The form is then the primary part. Otherwise, because it is a visible part, the part used for representing the data is the primary part. If parts are combined on a form, you should adjust their layout for the potential of resizing the window (see "Defining Visual Parts inside Other Visual Parts" in topic 1.4.3 for a more detailed description of parts to adjust to the size of the window).

3.1.3.2 Representation Part Checklist

The representation part should comply with the following characteristics:

- ☐ The representation part contains all parts that are visible to the user except the main set of buttons, the menu, and the window.
- ☐ The representation part does not contain data itself that is represented. It always points to data in the business object. The only exceptions are VisualAge Generator tables for lookup data and temporary variables that are not part of the business object.
- ☐ No user-initiated action that is and should be implemented as a push button or menu item is handled by the representation part.

+-----+ | GUI PROGRAMMING EXAMPLE: DEFINING VIEWS FOR A BUSINESS OBJECT +-----+

	<p>In this example we create two representations for the Baby business object. The first representation shows the detail information for the baby. The second representation shows the picture of the baby.</p> <ol style="list-style-type: none"> 1. Create a new GUI application. 2. Replace the window with a form. 3. Add a variable part to the free-form surface. Select Build Features Based on Member... and enter RBBB0WD. Make a Quick Form of the <u>self</u> attribute of the variable part on the form. Format and arrange the fields on the form. 4. Promote the <u>self</u> attribute of the variable part as <u>babyInformation</u>. 5. Add a second variable part to the free-form surface. Select Build Features Based on Member... and enter RBBB0WK. 6. Connect the <u>BABYID data</u> attribute of this variable part to <u>object</u> of the BABYID business object. 7. Promote the <u>self</u> attribute of this variable part as <u>babyKeyData</u>. 8. Save the GUI application as RBBBDDR.
--	---

3.1.4 Controls

A user can act on the business object by direct-manipulation techniques (drag-and-drop, double-clicking) or by using push buttons or items in the menu. The controls object contains the latter.

Each business object has its own controls object, although this can in turn consist of shared elements such as default menus or push button sets. There is one controls object per business object. The controls object is independent of the specific implementation of the representation. For example, it should be possible for the car salesperson to sell a car when showing the picture of the car or the detail technical information of the car.

Subtopics

3.1.4.1 Building the Controls Part

3.1.4.2 Controls Part Checklist

3.1.4.1 *Building the Controls Part*

The main components of the controls part are the window menu, object menu, and push buttons. Because the push button set is the only part of the controls part that has to be visible, it is the primary part. If it consists of multiple push buttons, these are placed on a form, and the form is made the primary part. If there are no push buttons, the primary part is a label.

Subtopics

- 3.1.4.1.1 The Window Menu
- 3.1.4.1.2 The Object Menu
- 3.1.4.1.3 Push Buttons
- 3.1.4.1.4 Security and Object States

3.1.4.1.1 The Window Menu

The window menu is usually the same for every business object. Every menu has **Edit**, **Windows**, and **Help** items. The CUA standards suggest a menu structure for an application whose user interface follows the object-oriented paradigm.

The standard menu structure can be built in an embeddable part of its own. One of the great features of a menu is that you can override any of the menu items by connecting another pop-up menu to the menu attribute. So if the standard menu is an embeddable part of its own, promoting the menu attributes of the different cascaded buttons in the menu allows you to override the menu items.

The control object embeds the standard menu. In the control object you place those menu items that you want to override.

You should also promote the label of the first menu item in the menu bar. When the application is run, this menu item should show the name of the business object with which the user is currently working. Connect a label with the name of the business object to the promoted feature to ensure this behavior.

The menu directs all user actions if it is present on the window. Otherwise all user actions are performed through push buttons. Therefore any action the user takes is redirected to the same action on the menu. This makes maintenance of the actions a lot easier.

For example, if your window has an **Open as details...** push button and an **Open as details...** menu item and will open the detail window when the user double-clicks on an entry in the list, these actions should all be directed to the menu item. The clicked event of the push button should be connected to the click event of the menu item. Likewise the defaultActionRequested of the list should be connected to the click action of the menu item. In this case the maintainability of the code outweighs the performance overhead of using clicked to click connections.

This choice of implementation means that you have to promote the click actions and clicked events of all menu items for which the reaction is implemented outside the controls or standard menu part.

3.1.4.1.2 The Object Menu

The object menu consists of three distinct parts, separated by a separator:

- ☐ Help and opening of business object views
- ☐ Access to the clipboard
- ☐ Object-specific actions

Each of these options is a subset of the regular menu; if possible, you should reuse these parts.

Object menus have a disadvantage in VisualAge Generator when used for a list. You cannot determine over which item in the list the user pressed mouse button 2 to get the object menu. Therefore you have to assume that it is the selected item in the list. This is contrary to the CUA guidelines. Therefore use object menus only in situations where the object menu relates directly to the object from which the user requested the object menu by using mouse button 2 (for example, when using a container of icons).

Note: You may be able to implement an Object menu with the appropriate selection support, using a container details part. We were not able to fully test this possibility.

3.1.4.1.3 Push Buttons

Push buttons are used to provide quick access for users to items that are also available on the menu. For example, in a data entry application, you can provide the user with a **Save and new** push button that has the default focus, thus allowing the user to enter data and either click the push button or press Enter to save it and get a cleared window to start entering the next set of data.

The push buttons should be made visible to the user. They are therefore part of the primary part of the controls part. The clicked event of the push buttons is connected to the click action of the related menu item.

3.1.4.1.4 *Security and Object States*

For the controls part you also have to consider security and the availability of menu items. Because the representation is affected by the security level of the user, any security mechanism should also be implemented in the controls part (see "Security" in topic 3.3.3.3.2 for a more detailed discussion of implementing a security mechanism in your GUI applications) and the standard menu.

Whenever a user is not allowed to access a certain action under his or her current security profile, and this is not dependent on the state of the object, the part should be destroyed. If a user is temporarily not allowed to access an action because of the state of the object, the action should be disabled.

These rules have an effect on the controls part. You should design for and implement states of the object. An object will have a finite number of states. Whenever the state changes, you have to check whether the change affects any of the parts that are used to represent or act on the business object. If so, the corresponding action should be taken.

3.1.4.2 Controls Part Checklist

The controls part should comply with the following characteristics:

- ☐ The controls part contains all of the push buttons and menus in the system. Exceptions are embeddable parts that are made part of the controls part or push buttons that are directly related to a certain field in the representation part.
- ☐ The menu window menu, if present, is the only part from which actions and events are promoted. If there is no window menu, events and actions are promoted from the push buttons.
- ☐ The controls part embeds the standard menu.
- ☐ The menus and push buttons should comply with the CUA standards.
- ☐ An object menu should only be used for objects that get focus when clicked on with mouse button 2.
- ☐ Object states and security should be implemented in the controls part and described in the design.

GUI PROGRAMMING EXAMPLE: CREATING A CONTROLS PART

In this example we create a simple controls part for the nursery application. The controls part contains a **Refresh** push button and a menu containing only the **Refresh** option as a choice in the **View** menu.

1. Create a new GUI application.
2. Replace the window with a **Refresh** push button.
3. Put a Popup Menu part on the free-form surface.
4. Promote the self attribute of this Popup Menu as babyMenu.
5. Drop a Menu Cascade part on the Popup Menu part. This will create a Menu Cascade part with a connection to a new Popup Menu part. Give the Menu Cascade part a label string of Refresh.
6. Drop a Menu Choice part on the Popup Menu part that is connected to the View Menu. Give the Menu Choice part a label string of Refresh.
7. Promote the clicked event of the **Refresh** Menu Choice as refresh. Connect the clicked of the push button to the click action of the **Refresh** Menu Choice.
8. Change the part names of all parts so they reflect your standards using the parts list.
9. Save the GUI application as **RBBDDC**.

3.1.5 *Common*

An application is not a stand-alone part in the entire application system. A lot of communication occurs between different parts of the system. All of the communication between different applications and business objects should be directed through the common part, for both the sender and receiver of information.

Each business object has its own common part. It is reused over different representations of the same business object.

Subtopics

3.1.5.1 Building the Common Part

3.1.5.2 Common Part Checklist

3.1.5.1 Building the Common Part

The common part is not visible to the user. It just provides a communication gateway to other applications in the system. Its primary part is therefore a label.

One form of communication that occurs between two applications is when one application requests the other application to be opened. For example, if the **Open as details...** menu item is clicked, the details window is opened. All activities associated with opening the other GUI application and the passing of data between them is part of the common part. The exact implementation depends on the mechanism you choose for opening other applications (see "Entry Point Application" in topic 3.3.3.3.1 for a more detailed discussion of opening other GUI applications).

Passing of data is an important aspect in this context, however. When you open a detail GUI application from a list application, you pass in data concerning the row that was selected in the list, the key. Because both the list and the detail application have their own business object, they both have their own working storage record containing the key information. Therefore you will have to pass data between the two parts. (16)

For performance reasons always pass data using an event that corresponds to the request for opening the window to the attribute that is to be set in the newly created window. The data attribute is the parameter to this connection. Always pass the data attribute of the working storage record, not any single data item of the working storage record. Although the key could just consist of one data item, you are not sure this will always be the case.

Note: There may be exceptions to this rule. You may need to expose level-77 data items as data triggers for application-specific goals. The idea behind always passing the data attribute of a working storage record is that modifications to the data items being exchanged do not require that the basic structure of connections and parameters be changed.

If you pass data using an event-to-attribute connection, you have to make sure that the target window is created before the connection is triggered. If the target window requires the data upon opening of the window, it should also be passed before the openWidget action.

If the current application is the target of communication, the common part contains the parts into which the data can be passed. These are always variable parts, of either working storage records or other parts in the system, such as the user part that can be used to implement security. Because they are variable parts their self attribute is promoted. For the variable parts that represent a working storage record in the business object of the application, the data attribute is also promoted.

(16) Passing self would cause ambiguity about the memory location of both working storage records.

3.1.5.2 Common Part Checklist

The common part should comply with the following characteristics:

- ☐ The common part only contains those parts that are part of the interface of the entire application.
- ☐ Other applications are opened from the common part. The exact implementation depends on your organization's implementation for opening other GUI applications.

Note: We have not defined a GUI programming example to build a common part. This part is not used in our nursery application.

3.1.6 *Interaction*

Once all of the parts have been created, they can be combined in a GUI application with a window as its primary part.

Subtopics

3.1.6.1 Combining the Parts in a GUI Application

3.1.6.2 Interaction Checklist

3.1.6.1 Combining the Parts in a GUI Application

The representation part and controls part are placed on the window. The common part is put in the bottom right-hand corner, and the business object is placed in the top right-hand corner. This layout gives the best results for the connections that need to be made between the parts and makes your applications easy to grasp quickly.

The following connections must be made between the parts:

☐ Business object to representation part

Connect the promoted data elements from the business object to the representation part. Because these will mostly be implemented (invisibly) as a connection between a working storage record and a variable part, the connections can be bidirectional. The connections should be easy to make if you have consistently given the characteristics that were identical and were promoted the same name.

Also connect any events of the business object that should cause refreshing of the data or validation to occur to the representation part.

☐ Business object to controls part

Connect the events from the controls part that should cause an action in the business object to each other.

☐ Business object to common part

Connect the promoted data elements from the business object to the common part. Because these will mostly be implemented (invisibly) as a connection between a working storage record and a variable part, the connections must be bidirectional. The connections should be easy to make if you have consistently given the characteristics that were identical and were promoted the same name.

☐ Representation part to controls part

Connect all events caused by direct manipulation from the representation part to the corresponding actions in the controls part.

Connect all events from the controls part that cause a change in the representation to the corresponding actions in the representation part.

Connect the object menus to the object menu handles provided by the representation.

☐ Common part to representation part

Connect any global parts such as a user part that is used to implement security from the common part to the representation part. The connection must be bidirectional because it involves variable parts.

☐ Controls part to common part

Connect all the events that should cause interaction with other applications to the actions that have been provided for this purpose in the common part.

Promote all features from the common part that need to be accessible by other applications to the interface of the GUI application. Examples are the data attributes of key working storage records and the attributes that indicate the global parts such as a user part.

Check the interfaces of the parts to see whether any features were promoted but not connected. Evaluate whether these features are required. Keep in mind that they can be used by other implementations of the same part.

3.1.6.2 Interaction Checklist

The combined parts should comply with the following characteristics:

- ☐ The combined parts should be embedded in a GUI application with a window as its primary part.
- ☐ The GUI application contains no other parts.
- ☐ The representation is placed on the window. The controls part is only placed on the window if the push buttons are to be visible to the user.
- ☐ The common part is placed in the bottom right-hand corner and the business object in the top right-hand corner of the free-form surface.
- ☐ The connections are made as described above.
- ☐ Check the relevance of the interfaces of the individual parts.

GUI PROGRAMMING EXAMPLE: PUTTING THE PARTS TOGETHER

In this example we combine the *baby* business object part (Creating a Business Object in topic 3.1.2.2), the representation part (Defining Views for a Business Object in topic 3.1.3.1) and the controls part (Defining Views for a Business Object in topic 3.1.3.2), in a GUI application. We did not implement a common part for this example.

1. Create a new GUI application.
2. Add the representation part (**RBBBDDR**) to the window.
3. Add the controls part (**RBBBDDC**) to the window. Arrange the parts as required.
4. Add the business object part (**RBBBZDB**) to the free-form surface
5. Make the following connections:
 - a. Representation part (**RBBBDDR**) babyKeyData and babyInformation attributes to the babyKeyData and babyInformation attributes of the business object part (**RBBBZDB**).
 - b. Controls part (**RBBBDDC**) babyMenu attribute to the menu attribute of the window.
 - c. Controls part (**RBBBDDC**) refresh event to the getBaby action of the business object part (**RBBBZDB**).
6. Test the application (save the member as **BABYBOY**).

When you click on the Refresh push button details for the baby identified in the entry field will be shown.

3.1.7 Application Development Considerations

The suggested application architecture provides you with maintainable and reusable applications. It still requires the discipline to apply the architectural requirements to the physical implementation of your GUI application systems. Changes to these suggestions are expected, but you should define a standard for organization and document any deviations from the standard that occur in any development activity.

Subtopics

3.1.7.1 Design

3.1.7.2 Documentation

3.1.7.1 Design

In applying the architecture to your application design, recognize the parts that will be the business objects and the representations that are possible on the business object. Several techniques can be used for determining the business objects in your system. You can take the Entity-Relationship (ER) model as your base, design the interface according to an object base paradigm (as suggested in the CUA standards), or design your system completely on the basis of business objects.

Your design should at least separate out the logical (business) parts from the representations on these parts. It should separate the user actions from the actions on the logical (business) parts.

The design should also include a definition of the states that the business object can be in, the actions that cause transitions between the states, and the consequences of each transition. This is not a requirement from an architecture perspective but from a perspective of building an event-driven application.

3.1.7.2 Documentation

Applications must be documented to allow for maintenance. Standardization is in this sense a form of documentation. It is preferable, though, to have other forms of documentation also.

The first form of documentation that should complement your application is the design. Because the application follows from the design, the documentation can also be used to analyze the application.

The second form of documentation is the technical design, where considerations made during the technical implementation of a certain business function are documented. In general, this kind of documentation is separated from the implementation in a separate document (which could have the same name with a different extension) or it can be included in the code.

For the latter option VisualAge Generator provides facilities for server applications, working storage records, VisualAge Generator tables, but not for GUI applications. GUI applications could be documented by including on the free-form surface a label explaining the working of the application. Because of the performance impact of this solution, it is not recommended.

An alternative that does not introduce a performance penalty would be to use an appropriately named external GUI application on the free-form surface and implement all documentation using label and multi-line edit parts. An extension to your naming standard could make this documentation external GUI application easy to identify and manage.

You can always choose to implement the technical design documentation in a separate document.

The third form of documentation is the technical documentation that explains the code. Again standardization helps a lot here. In a procedural language you can include comments. In a visual programming environment, you cannot. You can only use additional parts, such as labels, on the free-form surface.

A number of documentation facilities are available to you, though:

Printing a GUI application member

You can print a GUI application member. The printed member provides output detailing all parts that are included, all connections, and all features that were promoted. Here is an example of a printed member:

```
GUI APPLICATION NAME: MDL00LD
VisualAge Generator    DATE: 08-27-96    TIME: 02:48PM    PAGE: 00001
PROMOTED FEATURES

#models
  Part: 'models'
  Attribute: #self
#refreshList
  Part: 'pbRefreshList'
  Action: #enable
#pageInfo
  Part: 'pageInfo'
  Attribute: #self

COMPONENTS

FORM: frmModels
  CONTAINER DETAILS: cdvModels
    CONTAINER DETAILS COLUMN: colMake
    CONTAINER DETAILS COLUMN: colModel
VARIABLE: models
PUSH BUTTON: pbRefreshList ("refreshList" )
VARIABLE: pageInfo
VARIABLE: ROW of models
LABEL: Label1 ("Change History")

CONNECTIONS

Event-to-Action connection
  Event: 'pbRefreshList', #enabled
  Action: 'ROW of models', #getFieldsStartingAt:to:
  Parameter settings:
    #firstIndex : 1

Attribute-to-Attribute connection (single directional)
```

```
From: 'pageInfo', #ROWS-FETCHED data
To: ('pbRefreshList',#enabled -->
    'ROW of models',#getFieldsStartingAt:to:), #lastIndex
```

```
Attribute-to-Attribute connection (single directional)
From: ('pbRefreshList',#enabled -->
    'ROW of models',#getFieldsStartingAt:to:), #result
To: 'cdvModels', #items
```

```
Attribute-to-Attribute connection
From: 'models', #ROW
To: 'ROW of models', #self
```

To print the member to a file, set the print location in the **File** and **Print setup...** option to a filename. Information can be extracted from this file by using a REXX application. For more information about the use of printouts as a basis for tools around VisualAge Generator, see Appendix C, "VisualAge Generator Tooling" in topic C.0.

.ESF-based documents

A GUI application can be exported into a readable external source format (ESF) file. This format can be used to write a utility to extract some additional information. See Appendix C, "VisualAge Generator Tooling" in topic C.0 for more information about using .ESF files as a basis for tools around VisualAge Generator.

3.1.8 Summary

Application development requires applications that are built to comply to some form of standard architecture for the sake of maintainability and reusability of the code that is written. In this chapter we suggest such an architecture and provide you with the rules to which you should comply in building your applications.

The basis of the architecture is the fact that VisualAge Generator applications can be built up from parts. By using parts complexity can be hidden and each part can be made reusable. The architecture suggests that an application be built up from four basic parts (representation, business object, controls, and common), but these parts could actually contain other reusable parts.

Alternative architectural approaches are valid as long as they are consistently implemented, provide for reuse, and enable further extension of function within the boundaries of the application system.

Note: An example of an object-based approach to building VisualAge Generator application systems is provided in *Object-Based GUI Application Development with VisualGen*. Now that you have read this book, that example should actually make more sense.

The business object provides the interface to the business processes and transactions that you are trying to support in your applications. Often these have been built as server applications.

Each business object can be represented in a number of ways, for example, as a picture or textual information. The representation part is the implementation of one of these views on the business object.

User actions are represented through menu item or push button controls. The controls part includes these controls. Although certain actions will be available through both a menu item and a push button, the menu is leading in the implementation.

Applications communicate with each other. All communication, including the opening of other windows, is taken care of by the common part.

The parts are combined by embedding them in a GUI application. The representation part and controls part are put on the window. The other parts are placed on the free-form surface. By making the correct connections between the parts, you have a working application.

Building an application according to the architecture requires you to think through the process of going from a functional design to building an application. It also requires you to determine how the applications should be documented.

3.1.9 *What You Should Now Be Able to Do*

You should now be able to:

- ☐ Understand how to use an application architecture and its effect on your design process and documentation methods.
- ☐ Build an application according to the architecture.
- ☐ Check whether somebody else's application conforms to the application architecture.
- ☐ Understand the different possibilities provided by VisualAge Generator for documenting your system.

3.2 Chapter 15. Objects

This chapter examines the theory behind the structuring of parts. We look at why the architecture provides you with more maintainable and reusable code and talk about those aspects that you should consider in designing your system to accord with the architecture.

One of the main characteristics of the application architecture is that it hides complexity and allows for reuse of applications. The same two characteristics form the basis for object-oriented theory of designing application systems. In fact we have based the application architecture on object-oriented theory.

Subtopics

- 3.2.1 What Is an Object?
- 3.2.2 Object Interaction
- 3.2.3 Object Terminology
- 3.2.4 Designing for Objects
- 3.2.5 Summary
- 3.2.6 What You Should Now Be Able to Do

3.2.1 *What Is an Object?*

This section looks at the specific characteristics of objects and the effect of these characteristics on the application development process.

Subtopics

- 3.2.1.1 Real World
- 3.2.1.2 Abstraction
- 3.2.1.3 Encapsulation
- 3.2.1.4 Modularity
- 3.2.1.5 Hierarchy
- 3.2.1.6 Delegation
- 3.2.1.7 Persistence

3.2.1.1 Real World

An object is something from the real world. It is recognizable from the problem domain you are facing. In most cases you can touch it or at least describe it as a noun. Typical objects are airport, ATM, insurance, car, contract, house. The power is in the fact that you do not have to abstract far from your problem domain to start designing and programming a system. This reduces the chances of errors.

Looking at an object as something from our every day life, we can describe it as having:

☐ State

The state indicates the current condition of the object. The car is red. The contract has expired. These are all indications of the state of an object.

☐ Behavior

The state of the object can change. Changes are caused by the behavior of the object or by the behavior of other objects that influence the state of the object. The car is repainted by the garage. The contract is renewed under its own terms and conditions.

☐ Identity

Each object is unique. Your car differs from the car of your neighbor. If you have twins, each is unique, even if you have to identify them by putting red socks on one and blue socks on the other.

This base nature of an object and the fact that an object is something that is just out there in the world in which we live indicate that so far we have not lost touch with reality. The challenge is in describing these objects true to life.

3.2.1.2 Abstraction

Einstein's theory of relativity also has its impact on our description of the world. It is exactly that: our description of the world. We are describing our view of what these objects look like. In fact as application designers we have the even more difficult task of describing how somebody else, the user, looks at the world.

In determining how the world looks, our description should allow for only one explanation. It should contain the essential characteristics of the object we are trying to picture. Imagine we had to describe a mammal. What would be the essential characteristics of a mammal? The fact that it lives on the land? No. The fact that it gives birth from a womb? No.

Combined with a drive to describe an object in such a way that we can classify any object we encounter to be of a certain type, we have to look at the relevant perspective of the user. The biologist looks at the distinction between mammals and other animals in a different light than most of us do. The biologist also looks at the states and behaviors of living creatures differently than we do. Which view is correct? Neither. It depends on what you are trying to achieve.

Describing an object is therefore less straightforward than recognizing one. But describing an object correctly in light of the user's perspective enables us to ensure that the systems we build on this view of the world are correct.

3.2.1.3 Encapsulation

When interacting with an object, we generally do not want to know how the object performs its task. To be able to listen to your CD-player, you do not have to know how the technology is implemented that gets the data from the CD, transforms the digital signal to an analog signal, and comes out of your speaker. The implementation is hidden from view.

This concept, in object-oriented theory, is called *encapsulation*. It means that the complexity of the implementation is hidden from view by separating the interface from the implementation. You interact with the CD-player by pressing the **Play** button (the interface). This causes the laser signal to be evaluated and transmitted (the implementation).

Abstraction and encapsulation are closely related. The better your abstraction of a problem, the better the encapsulation generally is. A good abstraction provides a clear interface. A clear interface is an interface that does not contain anything that is not relevant to you.

3.2.1.4 *Modularity*

Objects can be grouped to form a module. This grouping of objects can in itself be seen as an object. As an object it has the same requirements. It should have clearly defined boundaries. As a general rule for grouping one could say that the communication of an object within its group must be more frequent than its communication over the boundary.

3.2.1.5 *Hierarchy*

One of the key concepts that object-oriented theory adds to the realm of application design is the concept of hierarchy. A hierarchy defines a ranking and ordering of objects, just like the one described in relation to the concept of modularity. This ranking and ordering can cause a relationship where one object is part of another object (aggregation) or where one object is a kind of another object (inheritance).

Subtopics

3.2.1.5.1 Aggregation

3.2.1.5.2 Inheritance

3.2.1.5.1 *Aggregation*

An object can itself consist of other objects. Just think of a car. It consists of wheels, a motor, and numerous other parts that are interrelated. Put all of these parts separately on the floor of your garage, and you do not have a car. You only have a potential for creating a car. Combined in a certain manner, the parts are a car.

Each part of the car is in itself an object and can again contain numerous other parts for which the same rule applies. If any of the parts does not comply with the standards, you either do not have a car or you have a bug (or, depending on your view of life, you have a feature).

3.2.1.5.2 Inheritance

One object can be a certain type of another object. Ford's first assembly-line-made car was available in only one model (for which you at least had a choice of color: black or black). Now each manufacturer has numerous models, but each of them is still a car. And as such, each of them has the base characteristics and behavior of a car.

This relationship can be seen as a hierarchy. A Ford Mustang is a kind of car, which is in turn a kind of vehicle. There are also other vehicles, such as boats. When trying to describe all vehicles, you could describe all of the characteristics and behaviors for each car and boat. A lot of characteristics and behavior are shared among the different vehicles, though.

If you start out describing all of the characteristics of a vehicle, and then those things that are different for cars followed by those aspects that are different for a certain model category, and so on, you would be writing a lot less.

If we could apply this same concept to our application development effort, we would also be writing a lot less code. It could increase your productivity enormously, provided of course that these kinds of structures are part of your problem domain. One of the defining characteristics of object-oriented programming languages is that they allow you to program and use the inheritance structure of a problem to reduce the amount of code you have to write.

3.2.1.6 Delegation

Another common design technique that allows objects to accept messages but not implement them directly is *delegation*. This means that an object can defer the processing of something to another object. This can be done, for example, by signaling some event from a control GUI application that tells the target GUI application to do some work. The control GUI application may even ask the parent GUI application to perform a specific operation that the control GUI application knows it cannot accomplish by itself.

This approach can impact system design and performance. You can share one instance of a complex object that implements a highly used function and let this instance know when you need some work done as opposed to embedding (duplicating) the complex object everywhere you need it.

This approach distinguishes delegation from aggregation; aggregation means you can use the same part in composing many different parts, but delegation allows you to use the one instance of the part from many other parts.

3.2.1.7 Persistence

A car exists as a car until you have it wrecked. The car is still a car if you are not driving it. Objects are persistent, meaning they exist even if they are not being used. Objects need to be explicitly destroyed.

In a development domain, this is a bit more difficult. The user can stop the application, shut down the machine, and go home. The object is gone from memory. Objects can be stored, however.

As objects consist of behavior and characteristics, storing an object would require you to store both these aspects. That is exactly what object-oriented databases do. But the behavior of an object rarely changes and is often implemented as part of an application, so only the characteristics need to be stored. Since characteristics are just like data, they can be stored in a relational database system or any other type of database system.

3.2.2 Object Interaction

Objects interact by sending each other messages. An object sends a message to another object when a certain event occurs. For example, a Wall Street broker has a number of stocks of a certain company in his or her portfolio. The stock is one of the objects. It has a number of characteristics, among which are the current notation and the level at which the stock should be sold.

When the price of the stock drops below the selling level, the state of the object changes. Its state now indicates "Get rid of this stock." This signal (event) can be used to trigger the selling action on the stock market.

In most situations the communication between the objects is basically one way. The target of the communication is not responsible for reacting. In fact, the target of the message might not even understand what the sender is talking about. For example, sending the sell stock message to a car does not make sense, and the car would certainly not understand what to do with the message.

Using events to trigger actions looks very much like programming in VisualAge Generator. In object-oriented languages, sending a message (performing an action) that the target does not understand either causes an error (runtime or compile time) or causes no action to be taken.

3.2.3 *Object Terminology*

The theory surrounding object orientation is loaded with terms that provide an unclear picture. The most evident of these is the word *object* itself.

Subtopics

3.2.3.1 The Word Object

3.2.3.2 Actions, Attributes, and Events

3.2.3.3 Class Tree

3.2.3.4 Private and Public

3.2.3.1 *The Word Object*

Object is in fact an instance of a class. A class is the definition of the behavior and the characteristics. Every instance behaves and has values for these characteristics.

For example, CD-player is a class. The class contains the definitions of the CD-player. It describes that a CD-player has a holder for a CD and can be played to produce signals that can be turned into sound. The CD-player also has a serial number and is a certain brand. Your CD-player has serial number "123." The class defines the fact that every CD-player has a serial number and that the serial number should comply with certain rules (numeric, less than eight digits). The serial number of my CD-player just has a value.

The class can be viewed as a template from which instances are created. Each instance obtains the behavior and characteristics of the class from which it is created.

3.2.3.2 *Actions, Attributes, and Events*

The interface of an object is defined in things it can do (behavior), the possible changes in its state, and its characteristics. These aspects have different names in different programming environments. Some programming environments call action methods or functions. Some programming environments do not recognize attributes because they can only be accessed by using actions. In this book we call the behavior of an object *actions*, the characteristics of an object *attributes*, and the change in the state of an object *events*.

3.2.3.3 Class Tree

The class tree is the term used for the inheritance hierarchy. The hierarchy not only saves you an amount of code you need to write because of the hierarchical relationship among the objects in the tree, it also allows a programming language that uses inheritance to seek the implementation of certain behavior up in the tree.

Imagine, for example, that you have defined a message window. The message window is a kind of modal window, which in turn is a kind of window. If we look at this from an inheritance perspective, all characteristics and behaviors that identify a window are defined for the window. For the modal window, only deviations are defined. The same is true for the message window.

If we could implement this structure, what would happen if we asked the message window for the title displayed in the title bar? This has not been defined as a characteristic of the message window because we have already defined it for all windows in general. A programming language that supports inheritance will recognize this relationship and seek up the hierarchy (up the tree) for the definition of the characteristic or behavior that is requested.

What if the described relationship exists but the message window's implementation of the title bar differs from the one defined for all windows in general. Because the definition for the message window is the first one that is encountered when going up the tree, it is used. For the person using the object, it is not relevant which of these situations occurs. The implementation is neatly encapsulated.

3.2.3.4 *Private and Public*

Behavior that is encapsulated and not part of the interface of an object is called *private*. You cannot request an object to perform this behavior. Object-oriented programming languages differ in their implementation concerning the default status of behavior when it is created. In some languages behavior is by default private unless you specifically make it public. In other languages it is just the other way around.

3.2.4 *Designing for Objects*

We have already been using some object-oriented concepts in VisualAge Generator without calling them objects. We called them parts.

If you build your application to make use of object-oriented concepts like inheritance, aggregation is not a requirement from the architecture perspective. When you build parts, you are automatically forced to clearly define what a part is, and what it is not. You are required to think through its interface and through the possible states it can be in. Recognizing the object-oriented relationships in building your parts can be very powerful, though. By using inheritance and aggregation, you can reduce the amount of code you write, make your applications perform better, and make your applications more maintainable.

If you are uncomfortable with object orientation, just design your parts and look for similarities. If you want to, you can tap into the full power of object-oriented techniques using VisualAge Generator. VisualAge Generator lets you encompass these new ideas at your own pace.

3.2.5 Summary

Object orientation is implicitly present in the application architecture. It allows reusable and maintainable code to be written.

An object is a recognizable and clearly definable "thing" in the real world. It has a state, behavior, and characteristics. Implementation of behavior is hidden inside the object. The interface defines what you can do with it.

Objects have relationships to each other. These relationships can have a certain ranking. Objects can be a kind of another object (inheritance), or they can be part of another object (aggregation). Objects are persistent. They exist while they are not in use. Database systems allow storage of the characteristics of objects.

Interactions between objects consist of messages. One object can send a message to another object when its state changes. The receiver of the message is not required to react but may do so.

The implications of object orientation for the design of your applications seem numerous. The most important aspect, though, is to recognize that objects are just like the embeddable parts you have been building. It is essential to clearly define the part, its boundaries, and interface. The fact that VisualAge Generator supports object orientation allows you to tap into the power of inheritance and aggregation.

3.2.6 What You Should Now Be Able to Do

This chapter will not have made you an expert in object orientation. It should, however, have given you a feel for the relationship and the applications you can build within VisualAge Generator. It should also have given you a reference point as to the difference between object orientation and the approaches you are used to.

You should now be able to:

- ☐ Understand the basic concepts surrounding objects and their relationships.

In the next chapter we provide more detail on how you can use VisualAge Generator to implement these concepts.

3.3 Chapter 16. *Objects in VisualAge Generator*

This chapter shows the implementation of the concepts described in Chapter 15, "Objects" in topic 3.2 within VisualAge Generator. It welcomes you to the power of using reusable structures in VisualAge Generator and covers the aspects that you need to consider to make your application work. It also discusses some of the possibilities that using objects in VisualAge Generator open up for you with regard to managing your GUIs and implementing security in your system.

First, we look at some of the aspects we have seen in other parts of this book and try to explain the behavior from an object-oriented standpoint. Next, we look at the characteristics of objects and how they can be implemented in VisualAge Generator. From this we distill some powerful possibilities.

Subtopics

- 3.3.1 VisualAge Generator Architecture
- 3.3.2 Building Your Own Objects
- 3.3.3 Opportunities
- 3.3.4 Summary
- 3.3.5 What You Should Now Be Able to Do

3.3.1 VisualAge Generator Architecture

The VisualAge Generator GUI builder and VisualAge for Smalltalk share a common code base. Both are based on Smalltalk. GUI applications are even generated as Smalltalk code. Smalltalk is an object-oriented language. Many of the classes provided in VisualAge for Smalltalk (the class tree) are also available in VisualAge Generator.

Subtopics

3.3.1.1 Parts Are Classes

3.3.1.2 Tear-off Indicates Aggregation

3.3.1.1 *Parts Are Classes*

In the VisualAge Generator GUI builder, the VisualAge for Smalltalk classes are actually the parts in the palette. Each class has actions, attributes, and events. These represent the behavior, characteristics, and identity of the class.

Any time you take a part from the palette and place it on the free-form surface, you are preparing your GUI application to create an instance of the class at runtime. Each class instance is unique and has a different name in the GUI application. This is true for all parts in the palette, except the working storage record, VisualAge Generator table, application, process, and statement group. If you change the name of these parts, they get a different definition. They are not classes.

The GUI application part is a class. You can embed multiple instances of a GUI application into another GUI application. If this is true, we are in fact saying that a GUI application is a class. Every GUI application you have built so far is in fact a class. It has an interface (the promoted features) and defines the behavior and characteristics of all the instances created from the class.

3.3.1.2 Tear-off Indicates Aggregation

When an attribute is torn off from a part, you get a new part with its own interface. For example, tearing off the iterator attribute of an ordered collection provides you with the possibility of looping through the array. Tearing off the label attribute of a label provides you with a part with the class EsString. This is in fact a collection of characters and has almost the identical interface of an ordered collection.

A part consists of other parts, just like a car consists of a motor, seats, and a steering wheel. Each part can be accessed individually, if it is part of the interface of the part. In fact, a tear off is a variable part. This is indicated by the bracket surrounding the part. It points to the attribute in the part from which it is torn off.

3.3.2 Building Your Own Objects

A GUI application is a class. This makes it possible for us to create classes. The parts in the architecture are in fact classes and have been designed as such. In Chapter 15, "Objects" in topic 3.2 we discuss the unique aspects of objects. Is it possible to implement these using VisualAge Generator? This section shows that it is and therefore VisualAge Generator holds within it an object-oriented programming tool.

Subtopics

- 3.3.2.1 Real World
- 3.3.2.2 Abstraction
- 3.3.2.3 Encapsulation
- 3.3.2.4 Aggregation
- 3.3.2.5 Inheritance
- 3.3.2.6 Communication

3.3.2.1 *Real World*

Objects are a representation of the real world. The business object used in the architecture is a representation of an object that is relevant for the business and therefore occurs in the real world.

3.3.2.2 *Abstraction*

Abstraction involves defining the boundary of an object. A GUI application allows you to define what the class stands for by defining its interface. The interface is the contract with other classes.

Abstraction is also about recognizing the different perspectives people have on an object. The architecture implements one business object but allows multiple representations to be built on top of this object, depending on the view and perspective of the user.

3.3.2.3 *Encapsulation*

The interface of the GUI application is separated from the implementation. If you change the implementation of a feature without changing the interface, you only have to regenerate the part, and not the applications in which it is embedded.

3.3.2.4 Aggregation

Aggregation is implemented in most parts in the product. You can also implement aggregation in your own classes (GUI applications). In fact the application architecture uses aggregation. The application consists of four objects: business object, representation, controls, and common. Building up an object from other objects is the same as embedding parts in a GUI application.

The parts that are embedded can be either private to the part in which they are contained or public. If they are public, they are accessible through the interface and can be torn off. If you promote the self of the embedded part, you can tear off the part from the part in which it is embedded and access all of its features.

GUI PROGRAMMING EXAMPLE: UNDERSTANDING AGGREGATION	
	This example clarifies the concept of aggregation. A good example of aggregation is representation that is a form that includes a number of entry fields. The representation consists of these aggregated entry fields.
	1. Create a GUI application. Replace the window with a form and place three entry fields on the form.
	2. Promote the <u>self</u> attribute of one of the entry fields. If you choose the first entry field, the default name is <u>text1</u> .
	3. Save the GUI application as AGGFORM .
	4. Embed the GUI application AGGFORM on a window of a new GUI application.
	5. Add a push button and an entry field to the window.
	6. Tear off the promoted attribute <u>text1</u> from the AGGFORM embedded GUI application.
	7. Connect the <u>clicked</u> event of the push button to the <u>backgroundColor</u> attribute of the new entry field off attribute. Provide the <u>object</u> attribute of the new entry field as the parameter.
	8. Test the application (save the GUI application as AGGWIN). The embedded part AGGFORM is an aggregation of entry fields. Because you promoted <u>self</u> , you can interact with one of the entry field subparts.

3.3.2.5 *Inheritance*

Relationships where one part is a type of another part occur a lot in application development using GUIs. A message window is a kind of modal window, which is in turn a kind of window.

Subtopics

3.3.2.5.1 Implementing Inheritance

3.3.2.5.2 Multiple Inheritance

3.3.2.5.3 Virtual Class

3.3.2.5.1 Implementing Inheritance

Inheritance in VisualAge Generator can be implemented by embedding the part from which you want to inherit in the part that inherits. It provides you with all of the function of the class that is embedded.

VisualAge Generator differs from other object-oriented programming languages in that, make use of inheritance, you have to explicitly promote all features of the embedded part (if the embedded part is not the primary part). The features of the part from which you inherit do not automatically become part of the interface of the part in which they are embedded. The features are by default private.

GUI PROGRAMMING EXAMPLE: IMPLEMENTING INHERITANCE FOR AN ORDERED COLLECTION

This example shows how you can inherit the behavior of an ordered collection to create a collection of whose contents can be reversed.

1. Create a new GUI application. Delete the window and replace it with a label. Promote the label with the text "reversibleCollection."
2. Put an ordered collection on the free-form surface. This is the part from which you inherit the behavior. Promote the features of the ordered collection to the interface with the same name as they have as part of the ordered collection. Because we want to promote all features of the ordered collection we need to promote all of the features because the ordered collection is not the primary part. Promote the self attribute of the ordered collection as orderedCollection.
3. Add a label to the free-form surface and provide it with the text "reverse." Promote the enable action of the label as reverse. We have created the base for an added feature of the ordered collection.
4. Add a working storage record named **RVOC-WS** with a the following structure:

10	SIZE	num	3
10	INDEX	num	3
77	TRIGGER	char	1
5. Connect the SIZE data attribute of the working storage record to the size attribute of the ordered collection.

.
.
.

.
.
.

6. Add a process named **RVOC-PROC** with the following code:

```
INDEX = INDEX + 1;  
IF INDEX LE SIZE;  
    TRIGGER = "Y";  
ELSE;  
    INDEX = 0;  
END;
```

7. Connect the enabled event of the label that causes the reverse to the execute action of the process.
8. Connect the TRIGGER data event of the working storage record to the removeAtIndex action of the ordered collection and provide the INDEX data attribute of the working storage record as the parameter.
9. Add the removed object back to the top of the ordered collection with these connections:
 - a. Connect the TRIGGER data event of the working storage record to the add:before action of the ordered collection
 - b. Connect the result attribute of the connection TRIGGER data to removeAtIndex newObject attribute of the connection just made.
 - c. Hardcode a value of 1 for the anIndex attribute of the connection to add:before

10. Connect the TRIGGER data event of the working storage record to the execute action process.
11. Save this GUI application as **RVOC**
12. Embed **RVOC** on the free-form surface of a new GUI application.

We can use the promoted features of the ordered collection or tear off the attribute ordered collection. If we tear off the attribute we can quickly form a listbox from the torn off ordered collection.
13. Create a listbox that will contain the contents of the ordered collection.
14. Add a push button and entry field to the window part. Implement connections that add the object of the entry field to the ordered collection.
15. Add another push button to the window part. Connect clicked of the push button to the reverse action of the embedded GUI application **RVOC**.
16. Test the application (save the member as **RVOCTEST**). You have created a part that does everything the base ordered collection part does and more.

3.3.2.5.2 *Multiple Inheritance*

Multiple inheritance allows a part to inherit behavior and characteristics from multiple other parts. A typical example is an amphibious vehicle that inherits the behavior of both a boat and a car.

Multiple inheritance requires that conflicts in the interface (both car and boat can have an action startEngine that differ) need to be resolved. If a language adds inherited features to the public interface by default, this can be difficult. Because VisualAge Generator defaults to private, you have the choice of choosing any implementation and even overriding it with your own.

3.3.2.5.3 Virtual Class

A virtual class is a class that has to be redefined for it to be used. You have to inherit from it to use its features. A virtual class is especially useful for visual parts that can contain other parts. If you make container details part with your own characteristics and embed this part elsewhere, you can no longer add columns to the container details.

A virtual class is a nonvisible class. Its major component is a variable part. You can act on the variable part just like you would on the part that is going to be used to redefine it. If that is a window, you can make a connection between the variable part's menu attribute (typing it as an unlisted feature) and a menu. This will give any window that inherits from this virtual class the same window.

To be able to use the virtual class you have to promote the self attribute of the variable part. When embedding, connect this attribute to the part that should inherit the characteristics and behavior.

GUI PROGRAMMING EXAMPLE: DEFINING A VIRTUAL CLASS	
	This example shows how you can use a virtual class to create a generic container details part.
	1. Create a new GUI application and replace the window with a label that indicates that the part will be a container details object.
	2. Put a variable part on the free-form surface and give it the name "containerDetailsView". Promote the <u>self</u> attribute of the variable part as <u>containerDetailsView</u> .
	3. Put another variable part on the free-form surface and give it the name "information". Promote its <u>self</u> attribute as <u>information</u> .
	4. Put a label on the free-form surface with the name "refreshList" and promote the <u>action</u> as <u>refreshList</u> .
	5. Connect the <u>enabled</u> event of the "refreshList" label to the <u>getFieldsStartingAt:toEndOf</u> of the "information" variable part (you must provide this as an unlisted action).
	6. Place two labels on the free-form surface. Give one a value of 1. Connect the <u>object</u> attribute of this label to the <u>firstIndex</u> attribute of the connection (first make a connection to <u>result</u> and replace it with <u>firstIndex</u> in the settings of the connection).
	7. Promote the <u>object</u> attribute of the second label as <u>numberOfRows</u> . Connect the <u>object</u> attribute of the label to the <u>lastIndex</u> attribute of the connection (first make a connection to <u>result</u> and replace it with <u>lastIndex</u> in the settings of the connection).
	8. Connect the <u>result</u> of the connection to the <u>items</u> attribute of the variable part that represents the container details.
	9. Save the GUI application as VRTCDV .
	.
	.
	.
	.
	.
	.
	.
	To use this virtual class we need to implement the part in a GUI application that represents container details. A quick example can be created using these steps:
	1. Create a new GUI application.
	2. Add a VisualAge Generator table to the free-form surface. The table should have a few columns and several rows of data. The MSGTBL created in Displaying Customized Messages topic 2.3.4.1.2 works very well. (We could also use a working storage record with data, but VisualAge Generator tables already have data.)
	3. Quick form the <u>table columns</u> attribute of the VisualAge Generator table to create container details on the window part. Delete the connection between the VisualAge Generator table and the container details.

4. Add the GUI application **VRTCDV** to the free-form surface.
5. Connect the container details self attribute to the containerDetailsView attribute of the **VRTCDV** GUI application.
6. Connect the table columns attribute of the VisualAge Generator table to the inform attribute of the **VRTCDV** GUI application.
7. Add an entry field and a push button to the window part. Connect the entry field attribute to the numberOfRows attribute of the **VRTCDV** GUI application. Connect the button clicked event to the refreshList action of the **VRTCDV** GUI application.
8. Test the application (save the member as **VRTCDVT**). You have made a generic solution for efficient loading of your container details that can be used by everyone in your organization.

3.3.2.6 *Communication*

Communication between objects takes place by sending messages. Within VisualAge Generator the connections that you create implement the message communication between objects. The connection is made between parts of the interface of both objects. In VisualAge Generator the receiver is not responsible for replying. The communication is unidirectional except for bidirectional attribute-to-attribute connections. Sending a message that is not understood by the receiver may cause a runtime error if the part is not a variable part.

3.3.3 *Opportunities*

The fact that VisualAge Generator fully supports object-oriented techniques creates a world of opportunities for your application.

Subtopics

3.3.3.1 Using the VisualAge Class Tree

3.3.3.2 Sharing Instances

3.3.3.3 Application Objects

3.3.3.1 Using the VisualAge Class Tree

In VisualAge Generator you typically just use the parts from the palette, your own parts, or tear-off attributes of parts. Because the base class tree of the VisualAge Generator GUI builder is almost identical to the base class tree of VisualAge for Smalltalk, however, you can also create some of the classes that are part of the VisualAge for Smalltalk product even though they are not exposed as part of VisualAge Generator.

We explain this using a sample. A function that is not available in VisualAge Generator but that you encounter now and again is the sort. In this example we create a sorting mechanism using classes that are available in VisualAge for Smalltalk. (17)

GUI PROGRAMMING EXAMPLE: ADDING VISUALAGE FOR SMALLTALK CLASSES TO A VISUALAGE GENERATOR GUI APPLICATION

An object factory can be used to create new parts at runtime. In fact the object factory creates new instances of a class. You provide the object factory with the name of the class and the attribute instanceClassName and when you perform the action new on the object factory, a new instance is created. We will use this function to pull classes out of the VisualAge GUI application programming environment that are not directly accessible:

1. Create a new GUI application and place an object factory on the free-form surface.
2. Provide "SortedCollection" as the instanceClassName. The sorted collection is a Smalltalk class that is just like an ordered collection except that the elements are automatically sorted (0-9, a-z).
3. Tear-off the instance attribute of the object factory.
4. Connect the aboutToOpenWidget event of the window to the new action of the object factory.
5. Put a list box, a push button, and an entry field on the window.
6. Connect the items attribute of the list box to the self attribute of the instance of the sorted collection.
7. Connect the clicked event of the push button to the add: action of the instance of the sorted collection. Provide the object attribute of the entry field as the parameter for the connection.
8. Test the application (save the member as **SORTOF**). The entries you add to the list should be sorted.

We do not intend to provide a Smalltalk reference here, but it may be worthwhile to take a look at the VisualAge for Smalltalk reference to find out more about the classes that are available. In VisualAge Generator not all of the classes are available nor are all of the features of the classes completely supported when they are available, but some provide you with potential benefit.

In the previous example we used an object factory to create a new instance of the class and work with that instance. There is another technique for introducing instances of a certain class into your application.

GUI PROGRAMMING EXAMPLE: EDITING EXTERNAL SOURCE FORMAT TO CHANGE A GUI APPLICATION

This example shows the use of the external source format of VisualAge Generator to create a class of a part.

1. Create a new GUI application. Place an ordered collection on the free-form surface, a list box, push button, and entry field on the window. Make sure that whenever the push button is clicked the text in the entry field is added to the ordered collection in the list box.
2. Save the GUI application as **SORTESF** and then export it to an .ESF file selecting **GUI application members in external format** toggle button.
3. Edit the .ESF file and replace the word "OrderedCollection" with "SortedCollection" (there are two instances).

		4. Save the .ESF file and import it into your current VisualAge Generator MSL, replacing the original GUI application.
		5. Test the application. The list should now be automatically sorted.
+-----	+-----	
+-----	+-----	

Note: The OrderedCollection really has become a SortedCollection. The features available for the part have changed. For instance, the `at:put:` feature, which is available for an OrderedCollection, is not available in a SortedCollection.

This use of VisualAge for Smalltalk classes opens up enormous possibilities. There are many useful classes that can be used. We mention only two:

Dictionary A dictionary is like an ordered collection but instead of being indexed by number it is indexed by the key you provide. You always have to add an item with its key.

Bag A collection of data with absolutely no ordering.

(17) Although the described techniques work, they are not documented and therefore not guaranteed to be supported in future releases of the product. Because of the shared code base between VisualAge for Smalltalk and VisualAge Generator, deviations seem unlikely, however.

3.3.3.2 *Sharing Instances*

In the application architecture we use a business object and a representation to view the business object. A business object can have multiple representations. For example, you can look at the list information of customers or a search criteria view on customers. Both, however, should logically use the same instance of the customer's object.

To be able to do so one of the applications could, instead of using the real business object, contain a pointer (variable part) to the business object. This way you only have one instance of the business object in your system. This would mean that the application that uses the variable part as the implementation can not be tested independently. This situation could be circumvented by using variable parts in both applications and providing a part that created the business object using an object factory when either of the applications is started, but uses the already created instance if it is already present.

When you frequently use the same business object in your application this technique:

- ☐ Reduces the number of parts in memory
- ☐ Ensures that data in each representation of the same business object is identical

This is a form of delegation (see "Delegation" in topic 3.2.1.6).

3.3.3.3 *Application Objects*

In Chapter 14, "Application Architecture" in topic 3.1 we only discuss the concept of a business object. In a real application there are a lot of other functions that need to be performed to provide the user with the required result. In this section we describe some of these functions and how the concept of objects can help us understand and implement them in our GUI application system.

Subtopics

3.3.3.3.1 Entry Point Application

3.3.3.3.2 Security

3.3.3.3.3 Application Management

3.3.3.3.1 Entry Point Application

An entry point application is the first application that users encounter when they start up the application. In a CUA-driven application with an object-oriented interface, this would include all the business objects the user can interact with. The entry point can be a container with icons, a set of push buttons, or a list with the object names. Preferably the icons, push buttons, or entries in the list are dynamic.

HINT

Icons and push buttons can be dynamically added to a part that can contain other parts. The subpartNamed:putOpened: range of actions. See "Adding Parts at Runtime" in topic for a more detailed description of these actions.

GUI PROGRAMMING EXAMPLE: IMPLEMENTING A SIMPLE ENTRY POINT APPLICATION

This example uses a list to implement a simple entry point application.

1. Create a new GUI application. Add a VisualAge Generator table, APPTAB, to the free-form surface and define it with the following structure:

```
10 APPNAME          char 8
10 APPDESC           char 50
```

2. Add a number of existing applications (external GUI applications in your current VisualAge Generator table as entries.

3. Tear off the table columns attribute of the VisualAge Generator table.

4. Put a list box on the window. Connect the APPDESC data attribute from the table tear-off to the items attribute of the list box.

5. Add an object factory to the free-form surface. Tear off the instance attribute of the object factory.

6. Add a working storage record named **APPWS** with the following structure to the free-form surface:

```
10 APPNAME          char 8
10 APPDESC           char 50
```

7. Add a process named **APP-PROC** with the following logic to the free-form surface:

```
RETR APPWS.APPDESC APPTAB.APPDESC APPWS.APPNAME APPNAME;
```

8. Connect the APPNAME data attribute of the working storage record to the instance attribute of the object factory.

9. Connect the selectedItem attribute of the list box to the APPDESC data attribute of the working storage record.

10. Connect the defaultActionRequested event of the list box to the execute action of the process, the new action of the object factory, and the openWidget action of the i

11. Test the application (save the member as **APPENT**). Double-clicking on an item in the list should start the corresponding application.

3.3.3.3.2 Security

A requirement for almost any application is the support for a security mechanism as dictated by the organization. The concept of objects facilitates an easy, natural implementation of security.

In a security mechanism there are two actors: the application and the user. Security is part of the description of their interaction. This interaction can take two general forms:

- ☐ The user determines which applications he or she is allowed to interact with and at what level.
- ☐ The application determines which users it is allowed to service and at what level.

From this perspective both the user and the application can be described as objects in the application system. The user provides the application with information about who the user is, what the security level of the user is, and depending on the implementation of security, even which applications (and parts of applications) the user is allowed to access.

Because security is proliferated throughout the application, the user object must be ever present. It can be initialized from an entry point application and passed around as a variable part.

At any point in the application, the part of the application can then:

- ☐ Provide its information to the user, at which point the user determines what he or she is and is not allowed to do
- or
- ☐ Require the user information from the user object, at which point the application determines what the user is and is not allowed to do.

The general rule for a user interface is that any actions that can never be taken by users with their current security clearance should not be present in the interface. Any actions that could be available dependent on the state of the application should be disabled.

Because a certain security state most likely has an effect on multiple aspects of the user interface, it is best to describe each of these possible states the application can be in. Each state should trigger the action to destroy or disable parts.

Because the GUI application itself is never 100% secure, you ought to consider rechecking the security level of the user in the server application.

3.3.3.3.3 *Application Management*

The interaction between GUI applications is an important part of the final business application. The application architecture indicates that this interaction should be part of the common part. Because GUI applications can also be dynamically created with an object factory, you could also envision a part that requires you to provide it with the name of the application that needs to be opened and takes care of the rest.

Using an object factory for this purpose reduces the interdependency between parts and increases the maintainability. It also reduces the amount of code that needs to be written in every GUI application.

3.3.4 Summary

VisualAge Generator supports object orientation. The parts in the parts palette and GUI applications are all classes from which instances can be created and placed inside a GUI application. The exceptions are the VisualAge Generator members excluding the GUI application.

VisualAge Generator supports all object-oriented concepts such as abstraction, encapsulation, aggregation, and inheritance. Aggregation is implemented by embedding a part in another part. Inheritance is also implemented by embedding a part in another part. Behavior and characteristics can even be inherited from multiple other objects. A class can also be defined such that it requires redefinition.

The fact that VisualAge Generator uses the same code base as VisualAge for Smalltalk allows you to create classes that are part of VisualAge for Smalltalk but are not included in the palette of VisualAge Generator. You can create them by using an object factory or by changing the class name of a part in an exported application.

Objects provide many new possibilities in VisualAge Generator. Object instances should only be created once in an application. The instance can be shared by all parts of the application that want to use the object.

Entry point applications, an implementation of security, and opening GUI applications from other GUI applications can all be made reusable structures more easily by using object orientation.

3.3.5 What You Should Now Be Able to Do

You should now be able to:

- ☐ Implement the different concepts of an object-oriented system, using VisualAge Generator.
- ☐ Use all of the object-oriented capabilities in VisualAge Generator to your advantage.
- ☐ Understand the benefits and possibilities of using an object-oriented paradigm in building your VisualAge Generator applications.
- ☐ Understand the relationship between the application architecture as described in Chapter 14, "Application Architecture" in topic 3.1 and the objects that support the architecture.

A.0 Appendix A. VisualAge Generator User Interface Standards

This appendix discusses the issue of user interface standards for GUI applications developed with VisualAge Generator. It describes only the suggested deviations from the guidelines set out in the CUA standards and highlights some specific compliances including their technical implementation in VisualAge Generator. Throughout this appendix we refer to the CUA reference manual and give the page numbers where you can find standards from which we suggest you deviate.

Subtopics

A.1 VisualAge Generator Limitations

A.2 National and Multiple Language Support

A.1 VisualAge Generator Limitations

VisualAge Generator does not support the following components of the CUA guidelines:

☐ Audible cue

Limitation: Cannot provide an audible cue when the user types a character for a mnemonic of a push button that is not part of the window (this is supported for the menu). This is only relevant if mnemonics are supported.

Solution: Either do not provide mnemonics or break this rule.

Limitation: VisualAge Generator does not give an audible cue when an item in a disabled state is selected (except when this item is a menu item).

Solution: Break this rule for nonmenu parts.

☐ Check box

Limitation: When using a check box to indicate the state of multiple objects, if some of them have the settings and others do not, you cannot set shade the check box.

Solution: Do not provide a check box to show the state over multiple objects.

☐ Choice

Limitation: The selection of an unavailable choice could provide the user with information about why the choice is unavailable in the information area of the window. This cannot be implemented in VisualAge Generator because there is no way of telling that a disabled item is selected.

Solution: This is not a firm requirement of CUA. Do not implement.

☐ Clear

Limitation: It is not possible to use the default delete folder of the operating system as the destination of the **Clear** choice.

Solution: This is not a firm requirement of CUA. Do not implement.

☐ Column heading

Limitation: VisualAge Generator GUI applications do not allow the user to edit the headings of a column themselves.

Solution: This is not a firm requirement of CUA. Do not implement.

☐ Contextual help

Limitation: Within VisualAge Generator there is no support for contextual help during a direct-manipulation operation.

Solution: This is not a firm requirement of CUA. Do not implement.

☐ Delete

Limitation: It is not possible to use the default delete folder of the operating system as the destination of the **Delete** choice.

Solution: This is not a firm requirement of CUA. Do not implement.

Limitation: The Delete key cannot be assigned as a shortcut key in VisualAge Generator.

Solution: Use Alt+Delete as the shortcut key for deletion.

☐ Delete folder

Limitation: It is not possible to use the default delete folder of the operating system.

Solution: Make your own delete folder or do not implement a delete folder in your application.

☐ Device

Limitation: VisualAge Generator does not contain functions for querying or using the available operating system devices.

Solution: This is not a firm requirement of CUA. If you want to use a "life" list of devices and use the device, however, build a program in another language or tool to provide this list and access the device.

□ Direct manipulation

Limitation: Within VisualAge Generator there is no support for contextual help during a direct-manipulation operation.

Solution: This is not a firm requirement of CUA. Do not implement.

Limitation: Drag and drop is not supported to target outside your VisualAge Generator application.

Solution: Only implement with your VisualAge Generator application. Use the clipboard for cross-application movement of data.

Limitation: VisualAge Generator does not show the part that is being dragged in shadowed when it is being copied versus full when it is being moved. Instead VisualAge Generator shows either a plus sign or no sign, indicating that the object is either copied or moved. Disadvantage is that you only see this when you move the object over a valid drop target.

Solution: This is not a firm requirement of CUA. Use as provided by VisualAge Generator.

□ Entry Field

Limitation: You cannot provide a visible cue to the user that the data in an entry field can be scrolled.

Solution: This is not a firm requirement of CUA. Do not implement.

□ First-letter cursor navigation

Limitation: This is not supported in VisualAge Generator for container details.

Solution: Build your own algorithm to determine the row to select or do not implement for a container details.

□ Folders

Limitation: VisualAge Generator has not implemented the concept of folders. It has a container part, but it is platform specific.

Solution: Do not design your product to work with folders.

Limitation: VisualAge Generator does not support objects in your application to be stored in the operating system folders.

Solution: This is not a firm requirement of CUA. Do not implement.

□ Help

Some functionality that is suggested should be part of the operating system provisions for help and the tool with which you build your help files. These aspects do not influence VisualAge Generator's capability to support CUA-compliant applications.

Limitation: VisualAge Generator does not provide a help facility on its message prompter part.

Solution: Do not implement yet or use your own message window.

□ Icon

Limitation: VisualAge Generator does not support the use of a small icon next to the system menu that can be the source or target for direct manipulation.

Solution: This is not a firm requirement of CUA. Do not implement.

□ Information area

Limitation: VisualAge Generator does not allow you to determine whether the user tried to select an object with unavailable-state emphasis. You can therefore not give a message about this fact in the information area.

Solution: This is not a firm requirement of CUA. Do not implement.

☐ Input focus

Limitation: VisualAge Generator does not support setting the input focus on the part that was nearest to where the user clicked. It requires this action to be exact.

Solution: This is not a firm requirement of CUA. Do not implement.

☐ Keyboard

Limitation: VisualAge Generator does not allow all interactions that are possible using the mouse to be performed using the keyboard, for instance, scrolling right in a container details that is read only.

Solution: Be aware of the limitations and try to limit the situations in which they occur.

☐ Message

Limitation: VisualAge Generator does not provide a help facility on its message prompter part.

Solution: Do not implement yet or use your own message window.

☐ Mnemonic

Limitation: VisualAge Generator cannot provide an audible cue when the user types a character for a mnemonic of a push button that is not part of the window (this is supported for the menu). This is only relevant if mnemonics are supported.

Solution: Either do not provide mnemonics or break this rule.

Limitation: The mnemonic should be reacted to independent of whether the upper-case or lower-case character is pressed. VisualAge Generator does not make this distinction for mnemonics of push buttons.

Solution: Wait for correction of this shortcoming of the product.

☐ Mouse

Limitation: VisualAge Generator does not allow all interactions that are possible using the mouse to be performed using the keyboard, for instance, scrolling right in a container details that is read only.

Solution: Be aware of the limitations and try to limit the situations in which they occur.

☐ Object

Limitation: Objects or GUI applications in VisualAge Generator can not be registered with the operating system.

Solution: All objects within your application system should be managed by your application system.

Limitation: VisualAge Generator does not allow objects within your application to support interaction with the standard objects provided by the operating system.

Solution: This is not a firm requirement of CUA. Do not implement.

☐ Open (Action Window)

Limitation: VisualAge Generator does not provide a help facility on its file prompter part.

Solution: Do not implement help yet on this part.

☐ Paste

Limitation: VisualAge Generator has no indicator whether the clipboard currently contains a value or not. You cannot disable the **Paste** option without pasting and checking the contents.

Solution: Provide a message when the user tries to paste while the clipboard is empty.

☐ Pointer

Limitation: VisualAge Generator does not show the pointer in a shadowed version when the object that is being dragged will be copied. Instead VisualAge Generator shows either a plus sign or no sign,

indicating that the object is either copied or moved. Disadvantage is that you only see this when you move the object over a valid drop target.

Solution: Use as provided by VisualAge Generator.

☐ Pointing device

Limitation: VisualAge Generator does not allow all interactions that are possible using the mouse to be performed using the keyboard, for instance, scrolling right in a container details that is read only.

Solution: Be aware of the limitations and try to limit the situations in which they occur.

☐ Pop-up menu

Limitation: Shift+F10 is not supported by VisualAge Generator as a shortcut key to the pop-up menu.

Solution: Be aware of the limitations and try to limit the situations in which they occur.

Limitation: A user has to explicitly select an object within a list to get the pop-up menu of the selected object.

Solution: Only provide pop-up menus where this limitation does not affect the behavior.

☐ Print

Limitation: VisualAge Generator does not contain functions for querying or using the available operating system devices.

Solution: This is not a firm requirement of CUA. If you want to use a "life" list of devices and use the device, however, build a program in another language or tool to provide this list and access the device.

Limitation: VisualAge Generator does not allow objects within your application to support interaction with the standard objects provided by the operating system.

Solution: This is not a firm requirement of CUA. Do not implement.

☐ Progress indicators

Limitation: You cannot show a progress indication on a server call because VisualAge Generator is single threaded, and server calls will not give back control to the calling GUI application until they are finished.

Solution: Provide a probable waiting time indication.

☐ Push button

Limitation: Esc is not supported as an accelerator for a Cancel push button.

Solution: Do not implement.

Limitation: The mnemonic of the push button should be reacted to independent of whether the upper-case or lower-case character is pressed. VisualAge Generator does not make this distinction.

Solution: Wait for correction of this shortcoming of the product.

☐ Radio button

Limitation: VisualAge Generator does not support mnemonics on radio buttons.

Solution: This is not a firm requirement of CUA. Do not implement.

☐ Reflection

Limitation: The concept of a reflection is not fully supported by VisualAge Generator.

Solution: This is not a firm requirement of CUA. Do not implement.

☐ Scroll bar

Limitation: VisualAge Generator does not allow scrolling right in a container details that is read only by using the keyboard.

Solution: Be aware of the limitations and try to limit the situations in which they occur.

□ Scrolling increment

Limitation: Scrolling increment is not supported on all list parts in VisualAge Generator.

Solution: This is not a firm requirement of CUA. Do not implement.

□ Shortcut keys

Limitation: No shortcut keys that are not combined with the Alt, Ctrl, or Shift are supported by VisualAge Generator.

Solution: Use alternative shortcut key combinations for those standard shortcut keys that are affected.

□ Source emphasis

Limitation: Source emphasis is not supported on direct manipulation with VisualAge Generator.

Solution: Implement direct manipulation without source emphasis.

□ Split

Limitation: VisualAge Generator does not support windows to be split into panes that can be manipulated by the user. It does not support addition of features to the system menu.

Solution: Do not provide windows that have to split. Use two windows instead or present the data in a container details and use the lockedColumns attribute to show a certain number of the leftmost columns.

□ Split window

See *Split*

□ Target emphasis

Limitation: Target emphasis is not supported on direct manipulation with VisualAge Generator.

Solution: Implement direct manipulation without target emphasis.

□ Tool palette

Limitation: VisualAge Generator does not allow you to change the mouse pointer to indicate the tool that was chosen.

Solution: Implement without this change in mouse pointer function.

□ Unavailable-state emphasis

Limitation: VisualAge Generator does not allow you to determine whether the user tried to select an object with unavailable-state emphasis. You can therefore not give a message about this fact in the information area.

Solution: This is not a firm requirement of CUA. Do not implement.

Limitation: VisualAge Generator does not give an audible cue when an item in a disabled state is selected (except when this item is a menu item).

Solution: Break this rule for nonmenu parts.

□ Undo and redo

Limitation: Undo and redo cannot always be provided in a multiuser, online system that stores data in a file system or database.

Solution: Do not implement undo and redo in these cases.

□ Value set

Limitation: The mnemonic of the value set push button should be reacted to independent of whether the upper-case or lower-case character is pressed. VisualAge Generator does not make this distinction.

Solution: Wait for correction of this shortcoming of the product.

☐ Window layout

Limitation: Providing the user with a choice of clipping or resizing window contents upon sizing of the window is not easily supported by VisualAge Generator.

Solution: This is not a firm requirement of CUA. Do not implement.

☐ Window navigation

Limitation: Not all parts support mnemonics in VisualAge Generator.

Solution: This is not a firm requirement of CUA. Do not implement.

☐ Work area

Limitation: VisualAge Generator has not implemented the concept of work areas. It has a container part, but it is platform specific.

Solution: Do not design your product to work with work areas.

Limitation: VisualAge Generator does not support objects in your application to be stored in the operating system work areas.

Solution: This is not a firm requirement of CUA. Do not implement.

A.2 National and Multiple Language Support

If your application is in a language other than English, use the translated terms as suggested by CUA for any interface-related terms.

Be aware of the fact that standard dialogs that can be used in your VisualAge Generator GUI applications are presented to you in the language of your operating system.

If your application needs to support multiple languages at runtime, in other words the user can choose the language in which to work with the application, the following considerations apply:

- ☐ Do not use the standard dialog boxes provided by the operating system. These will show text in the language of the operating system.
- ☐ Be aware of the fact that the length of labels and entry fields in different languages should be different and that your window should automatically resize to adjust for these differences.
- ☐ Right-to-left languages should have separate windows designed for them because they need to be mirrored over the y-axis of the window. VisualAge Generator does not support such automatic mirroring.

B.0 Appendix B. VisualAge Generator Naming Convention

This appendix discusses the issue of naming standards in a VisualAge Generator application development environment and defines the naming standards and guidelines used for VisualAge Generator members and parts.

By reviewing the necessary considerations in giving names to VisualAge Generator members, a logical, consistent naming convention is proposed. The naming standard encompasses names of VisualAge Generator members, such as applications and data items, and GUI parts, such as push buttons.

The given standard is a suggestion. We indicate possible deviations from this standard to make the convention applicable to your situation.

Subtopics

B.1 Considerations

B.2 Member Names

B.1 Considerations

A number of technical and organizational aspects influence the names that can be used for VisualAge Generator members.

Subtopics

- B.1.1 Requirements
- B.1.2 Uniqueness
- B.1.3 Meaningfulness
- B.1.4 Existing Standards
- B.1.5 Member List
- B.1.6 Development Paradigm
- B.1.7 Platform Independence
- B.1.8 Aliases
- B.1.9 CICS Transaction ID

B.1.1 Requirements

The naming convention must comply with the allowable lengths of VisualAge Generator members and parts and with the restriction in the use of special characters. Refer to the VisualAge Generator documentation for a summary of these restrictions.

B.1.2 Uniqueness

Member names must be unique both within and outside its type. For example, having a process and an application called ACLDDDP is not allowed in VisualAge Generator.

B.1.3 Meaningfulness

Development activity is assisted when names are meaningful, predictable, and contain some information relative to the purpose of the member. An edit process might be called EDIT, but this name might not be valid because there is no ownership information (which application uses the process) or reference to what type of edit the process performs. Meaningfulness also increases the chances of reuse of the application component.

B.1.4 Existing Standards

Coexistence or correspondence to existing standards might be of importance in your organization (for example, to preserve uniqueness of application names or ease migration). The suggested naming standard is based on the European IBM CSP naming standard. It has been enhanced to include the specific features of VisualAge Generator.

B.1.5 Member List

The VisualAge Generator member list allows the user to sort and view the member in a concatenation of MSLs, from different perspectives. Search criteria can be set using wildcards on any character of the member name, and the list can be ordered by name, type, or the update time stamp. To be able to use the member list effectively the positions in the member names should have a consistent meaning. This will allow for accurate searches and a meaningful ordering of the members on basis of the name.

B.1.6 Development Paradigm

One of the main considerations in a naming standard is the paradigm used during development. For example, is the application object oriented or application oriented? Are objects reused over the different application systems that are built? Is user interaction based on first choosing the object followed by the action, or the other way around?

The suggested naming standard is based on an object-oriented view of an application system. It assumes object-action-based user interaction. Although objects instead of applications form the basis of the naming standard, the standard is by no means limited to systems designed using object oriented techniques.

B.1.7 Platform Independence

One of the key features of VisualAge Generator is the ability to build platform independent code. This should also be reflected in the naming standard. For example, working storage records that reflect data in a database should not blindly follow the name of the table in the relational database (although this might be appropriate for an SQL record). The name of the table or the type of database system might change and this would remove the meaning from the name of the working storage record.

B.1.8 Aliases

To avoid aliases being assigned during COBOL or C++ generation and to improve the readability of the generated COBOL or C++ program, these standards should be followed:

- ☐ Use 30 characters or less in item names.
- ☐ Do not use COBOL or C++ reserved words.
- ☐ Do not use \$, #, @, or _ characters.
- ☐ Do not use double-byte character set (DBCS) names for item names if our application contains SQL processes.
- ☐ For C++ applications, do not use DBCS names.

B.1.9 CICS Transaction ID

VisualAge Generator uses the first four characters of the application name to generate the default CICS transaction ID. This ID can be overruled during generation, but a useful initial value can be beneficial to the generation process.

B.2 Member Names

The naming conventions for applications developed with VisualAge Generator have the format shown in Table 13:

Table 13. VisualAge Generator Member Names.								
1	2	3	4	5	6	7	8	VisualAge Generator Object
b	b	o	o	v	U	P		Umbrella application
b	b	o	o	w	A	P		Atomic application
b	b	o	o	w	A	P	x	PSB
b	b	o	o	w	T	x		VisualAge Generator table
b	b	o	o	w	M			Map group
b	b	o	o	v	M	x	x	Map
b	b	o	o	w	R	x	x	Record
b	b	o	o	w	F	x	x	File or DL/1 data
b	b	o	o	w	H			Help map group
b	b	o	o	w	H	x	x	Help map
b	b	o	o	w	P	z	z	Process
b	b	o	o	w	S	z	z	Statement group
b	b	o	o	w	W	z	z	Working storage
b	b	o	o	w	W	R	z	Redefined working storage
b	b	o	o	v	c	P		GUI application
b	b	o	o	v	c	t		Part in application architecture
X	b	b	o	o	w	A	x	Exported atomic application
X	b	b	o	o	w	U	x	Exported umbrella application
X	b	b	o	o	v	G	x	Exported GUI application

Where:

bb	Business identifier (for example, AC for accounting). Because parts built with VisualAge Generator can be reused by different systems, the business identifier is not required, although you may have to include one to comply with your organization's existing naming standard. If a business identifier is not required, these characters can be used as extra indicators of the object.
oo	Object identifier (for example, LD for ledger). First four characters if no business identifier is used.
v	Type of representation (photo, detail) with which the part is associated; a Z when the part is independent of a representation.
w	A unique identifier of the usage of the application or part which you may want to relate to its function (save, delete, select).
c	Indicates whether this is a container of other objects or a single object.
t	Object type in the application architecture.
B	business object
R	representation part
C	controls part
X	common part
M	object menu
x	A unique numeric identifier for the member.
z	A meaningful addition to the name within the boundaries of

the rules for naming members.

Possible deviations to fit the standard of your organization are:

- ☐ Include the member type in the name.
- ☐ Use the eighth character of the GUI application names for additional information.
- ☐ Reorder the meaning of the characters.
- ☐ Use only three characters as the business and object identifier and use the additional character for another purpose.

Subtopics

B.2.1 Descriptions

B.2.2 Records and Tables

B.2.3 GUI Names

B.2.1 Descriptions

On a number of VisualAge Generator members, several characters are available as a suffix to give a further description of the member. The following uses of these descriptions are suggested:

□ Execute processes and statement groups

A meaningful suffix is added to the process or statement group separated by a hyphen (-). The following uses are suggested:

MAIN for the main process

INIT for initialization of the application

PARMS for setting of the application specific parameters

BL-DESC for business logic

Example: ACLD0PU-MAIN

□ Processes with an I/O process option

These processes will have a suffix indicating the process option used separated by a hyphen (-). This suffix can be enhanced with a further distinction (for instance, recognition of forward and backward scrolling). Finally, a business logic indication can be added (BLB - business logic before; BLA - business logic after).

Example: ACLD0PA-SIQ-FW-BLA.

Table 14 shows the abbreviations of process options.

Table 14. Abbreviations of Process Options.	
Process Option	Abbreviation
ADD	ADD
CLOSE	CLS
CONVERSE	CNV
DELETE	DEL
DISPLAY	DSP
EXECUTE	(none)
INQUIRY	INQ
REPLACE	REP
SCAN	SCN
SCANBACK	SCB
SETINQ	SIQ
SETUPD	SUD
SQLEXEC	SQX
UPDATE	UPD

B.2.2 Records and Tables

Names for records and tables are discussed in this section.

Subtopics

B.2.2.1 SQL records

B.2.2.2 Records

B.2.2.3 Data Items

B.2.2.4 Common Records

B.2.2.1 SQL records

SQL records are records directly related to a table or view in a relational database system. The name of the SQL record definition should be the same as the relational database table or view name they represent. Alternative names may be used when multiple SQL row records are created for the same table to provide alternative selection criteria. This should be done by providing a meaningful suffix to the original name.

B.2.2.2 Records

A number of general record structures can be recognized that are useful in building an application. The following meaningful additions are made to the working storage record names to make them easy to identify and use:

- ☐ bboowWL
Contains the result of the fetched data for the list (array).
- ☐ bboowWD
Contains all of the data of one row for the details window.
- ☐ bboowWK
Contains the key data items for the record.
- ☐ bboowWC
Contains all of the input data on the user's search screen. These values are used for the search. The occurs is 1 and is the communication between the client and the server.
- ☐ bboowWV
Contains the controls for the search criteria values. It is identical to the above mentioned record, except that every data item occurs twice. The items are used to store the high and low values used during searching. This record is only used within the atomic server application.
- ☐ bboowWS
Contains the controls for the values for scrolling (upper and lower limits).

When redefinitions are made (for instance, not all data is shown in a specific list), these can be indicated using a redefinition sequence number or an alphanumeric addition that makes the name meaningful (for instance, ACLDOWL-NAME).

B.2.2.3 Data Items

Data items that correspond to columns in a database system use the same names as the columns they represent. These data items should be declared as global. Local data items should follow the naming standard for data item members.

B.2.2.4 Common Records

Within the organization a number of common records can be defined that contain information that needs to be shared between applications. These records can be named as you see fit, although consistency and recognizability are suggested. The records should be standardized on an organizational level.

B.2.3 GUI Names

Names for GUI application parts and promoted features are discussed in this section.

Subtopics

B.2.3.1 Part Names

B.2.3.2 Promoted Features

B.2.3.1 Part Names

A GUI part is a self-contained software component with a public interface consisting of a set of external features that enable the part to interact with other parts (actions, attributes, and events).

When you add parts to the free-form surface or to the windows on the free-form surface, the VisualAge Generator GUI Definition Facility automatically assigns names to the part, according to the part type and how many other parts of that part type exist in the GUI application. You may have a considerable number of parts from the same category, such as push buttons or data entry fields. As you modify and enhance the GUI application, it becomes difficult to distinguish among the different parts. We suggest that you use names like pbXXXXx (pbHelp).

Table 15 shows the part naming guidelines:

Table 15. GUI Part Naming Guidelines. The names shown guided our development activity. You should define a naming guideline for your environment that covers all GUI application part categories you use in your applications.			
Part Category	Part Type	VisualAge Generator Default Name	Suggested Name
Button Category	Push Button	Push Buttonx	pbXXXXx
	Toggle Button	Toggle Buttonx	tbXXXXx
	Radio Button	Radio Button Setx	rbXXXXx
	Scale	Scalex	scXXXXx
	Slider	Sliderx	slXXXXx
	Hot Spot	Hot Spotx	hsXXXXx
Data Entry Category	Text	Textx	efXXXXx
	Formatted Text	Formatted Textx	ftXXXXx
	Multi-line Edit	Multi-line Editx	mleXXXXx
	Table	Tablex	taXXXXx
	Label	Labelx	laXXXXx
	Spin Button	Spin Buttonx	sbXXXXx
List Category	List	Listx	liXXXXx
	Multiple Select List	Multiple Select Listx	mslXXXXx
	Drop-down List	Drop-down Listx	ddlXXXXx
	Combo Box	Combo Boxn	cbXXXXx
	Spin Button	Spin Buttonx	sbXXXXx
	Container Details View	Container Details Viewx	cdvXXXXx
	Container Details Column	Container Details Columnx	cdcXXXXx
Menu Category	MenuBar	MenuBarx	mbXXXXx
	MenuBar Item	MenuBar Itemx	mbiXXXXx
	Popup Menu	Popup Menux	pmXXXXx
	Menu Choice	Menu Choicex	mcXXXXx
	Menu Cascade	Menu Cascadex	caXXXXx
	Menu Toggle	Menu Togglex	tbXXXXx
	Separator	Separatorx	seXXXXx
	Window	Windowx	winXXXXx
	Form	Formx	foXXXXx

VisualAge Generator GUI Development Guide

Part Names

Canvas Category	Group Box	Group Boxn	gbXXXXX
	Scrolled Window	Scrolled Windowx	swXXXXX
	PM Notebook	PM Notebookx	pnbXXXXX
	Windows Notebook	Windows Notebookx	wnbXXXXX
	Notebook Page	Notebook Pagex	nbpXXXXX
Prompters	Message Prompter	Message Prompterx	mpXXXXX
	Text Prompter	Text Prompterx	tpXXXXX
	File Selection Prompter	File Selection Prompterx	fspXXXXX
Models	Ordered Collection	Ordered Collectionx	ocXXXXX
	Object Factory	Object Factoryx	ofXXXXX
	Variable	Variablex	varXXXXX
	File Accessor	File Accessorx	faXXXXX
OS/2 Category	Container	Containerx	conXXXXX
	OS/2-Windows Notebook	OS/2-Windows Notebook1	nbXXXXX
DDE	DDE Client	DDE Clientx	ddcXXXXX
	DDE Server	DDE Serverx	ddsXXXXX
Multimedia (parts)	Compact Disc Player	Compact Disc Playerx	cdpXXXXX
	Video Disc Player	Video Disc Player	vdpXXXXX
	Audio Wave Player	Audio Wave Playerx	awpXXXXX
	Digital Video Player	Digital Video Playerx	dvpXXXXX
	Video Playback Window	Video Playback Windowx	vpwXXXXX
Multimedia (buttons)	Motion Buttons	Motion Buttonsx	mmbXXXXX
	Track Buttons	Track Buttonsx	mtbXXXXX
	Frame Buttons	Frame Buttonsx	mfbXXXXX
	Record Button	Record Buttonx	mrBXXXXX
	Eject Button	Eject Buttonx	mebXXXXX
	Wrap Button	Wrap Buttonx	mwbXXXXX
	Mute Button	Mute Buttonx	mmuXXXXX
	Play Button	Play Buttonx	mpbXXXXX
	Stop Button	Stop Buttonx	msbXXXXX
	Rewind Button	Rewind Buttonx	mrwXXXXX
	Fast Forward Button	Fast Forward Buttonx	mffXXXXX
	Pause Button	Pause Buttonx	mpsXXXXX
	Frame Reverse Button	Frame Reverse Buttonx	mfrXXXXX
	Frame Advance Button	Frame Advance Buttonx	mfaXXXXX
	Track Reverse Button	Track Reverse Buttonx	mtrXXXXX
	Track Advance Button	Track Advance Buttonx	mtaXXXXX

Using these names makes the connection descriptions shown at the bottom of the window and in the trace more readable.

B.2.3.2 Promoted Features

The public interface of parts in VisualAge Generator should use the following naming convention:

☐ Actions

Actions are stated as verbs. If the action requires parameters, a colon delimits the words indicating the type of parameter expected. The first word is lower case as is the first word after a colon. All other words start with an upper case letter (for instance, getFieldsStartingAt&:.to&:.). Remember no spaces are allowed within the names of actions.

☐ Attributes

Attributes are stated as nouns. Often the attributes are representations of values in a working storage record and might represent either the self or the data attribute of the data item in the working storage record. The self attributes are stated as a description of the data item (for instance, ACLDOWK will be modelKey), and the data attributes have an additional suffix of Data (thus: modelKeyData). The nouns are capitalized like the actions. Although spaces are allowed in the names of attributes, we suggest you not use them.

☐ Events

Events are stated as a past-perfect participle. They should indicate the action that has just finished. The first word is lower case (for example, listRefreshed). Although spaces are allowed in the names of attributes, we suggest you not use them.

C.0 Appendix C. VisualAge Generator Tooling

VisualAge Generator provides a number of external interfaces that you can use to build tools around the product enhance its usability. This chapter describes these handles and their possible uses.

Subtopics

C.1 External Source Format

C.2 Print File

C.3 Summary

C.4 What You Should Now Be Able to Do

C.1 External Source Format

You can export a VisualAge Generator GUI application into .ESF by selecting **Utilities** and **Export** from the menu when you have selected the GUI application in a member list. By selecting **Export GUI members in external format**, the .ESF for the GUI application is in a readable format. The format looks a lot like Smalltalk (actually, it **IS** Smalltalk).

The generic structure of the .ESF is:

```
:guiapp
.
.
.
:script
.
.
.
:escript
.
.
.
:eguiapp
```

The definition of the GUI application is contained between the script tags. The definition consists of three sections in the following sequence:

Internal structure

This section of the .ESF file (see Figure 56) identifies the parts that are to be created as part of the GUI application. This part of the file is identified by the word `calculatePartBuilder (1A)`.

```
+-----+
|
| !MEMBER class publicMethods ! ( 1B )
|
| calculatePartBuilder ( 1A )
|   "Create the edit time part tree for the MEMBER."
|   " MEMBER recalculatePartBuilderRecordFromArchivalCode. "
|
|   ! aMEMBER ! ( 1C )
|
|   aMEMBER := self newTopLevelPartBuilder.
|
|   aMEMBER ( 1D )
|     subpartBuilderNamed: 'partName'
|       put: (self addClassNamePartBuilder: aMEMBER);
|       :
|       :
|     subpartBuilderNamed: 'tearOffName'
|       put: ((AbtTearOffAttributeSpec new attributeName:
|         'attributeName') newVariableBuilder);
|   aMEMBER ( 1E )
|     attributeSettingNamed: #primaryPart put:
|       (aMEMBER subpartBuilderNamed: 'primaryPartName').
|
|   self connectMEMBER: aMEMBER. ( 1F )
|
|   !aMEMBER! ( 1G )
|
+-----+
```

Figure 56. Internal Structure Section of an ESF File for a GUI Application

Although this may look like incomprehensible code, it can be read as a sequence of statements.

This is the definition of the public methods of the MEMBER class (1B). One of these is the calculatePartBuilder method. It defines how the part should be built. The method uses one variable, aMEMBER (1C), which is assigned to a newly created top-level part (the GUI application).

For a MEMBER it then executes a number of statements that build the parts that are embedded in the top-level part aMEMBER (1D). It sends the subpartBuilderNamed:put: message to the aMEMBER variable and indicates the name of the part to be created and the method by which it should be created. The implementation of these methods is stated at a later point in the .ESF. The same kind of structure applies to tear-offs.

It then indicates the primary part of the application (1E) and calls a method that creates all the connections (see 1F in Figure 56 and Figure 57).

Finally it returns aMEMBER (1G) to whomever calls this method.

```
connectMEMBER: aTopLevelPartBuilder ( 1F )
    "Create the edit time part tree for the MEMBER."

    aTopLevelPartBuilder
        connectionBuilderAt: 0
            put: (AbtAttributeToAttributeConnectionBuilder new
                sourceBuilder: (aTopLevelPartBuilder subpartBuilderN|med:
                    'sourcePartName');
                sourceAttributeName: #sourceAttributeName;
                targetBuilder: (aTopLevelPartBuilder subpartBuilderN|med:
                    'targetPartName');
                targetAttributeName: #targetAttributeName);
```

Figure 57. Method to Create Connections from an ESF File for a GUI Application

Visual layout

This section (see Figure 58) defines the placement, connections, and initial attribute values of parts on the free-form surface. AbtShellView is the VisualAge Smalltalk name for the window part.

```
calculateVisualLayoutPartBuilder ( 2A )
    "Create the edit time part tree for the AbtShellView."
    " AbtShellView recalculatePartBuilderRecordFromArchivalCode. "

    | anAbtShellView anAbtVariableIconView anAbtCwActionConnectionView
      anAbtCwConnectionView |

    anAbtShellView := AbtShellView newSubpartBuilder. ( 2B )

    anAbtShellView
        subpartBuilderNamed: 'partName'
            put: (anAbtVariableIconView := AbtVariableIconView newSubpartBuilder);
            :
            :

    ( 2C )      subpartBuilderNamed: ' ('sourcePartName',#event -->
                'targetPartName ',#action)1'
                put: (anAbtCwActionConnectionView := AbtCwActionConnecti|nView
                    newSubpartBuilder).

    anAbtShellView
        attributeSettingNamed: #initWidgetSize put:
            (Rectangle origin: (Point x: 0 y: 38) corner: (Point x: 40 y: 480)).

    anAbtVariableIconView
        attributeSettingNamed: #graphicsDescriptor put: (AbtIconDesc|riptor new
            moduleName: 'AbtBmp##'; id: 903; shading: AbtNormalGraph|c);
        attributeSettingNamed: #framingSpec put: (AbtViewAttachments|ec new
            leftEdge: (AbtEdgeConstant new
                offset: 356);
            rightEdge: (AbtEdgeConstant new
                offset: 54);
            topEdge: (AbtEdgeConstant new
                offset: 246);
            bottomEdge: (AbtEdgeConstant new
                offset: 65));
        attributeSettingNamed: #label put: 'employees';
        :
        :

    anAbtCwConnectionView ( 2D )
        attributeSettingNamed: #midPoints put: ((Array basicNew: 1)
            at: 1 put: (Point x: 491 y: 340);
            yourself).

    |anAbtShellView! !
```

Figure 58. Visual Layout Section of an ESF File for a GUI Application

This is the calculateVisualLayoutPartBuilder method (2A). It uses a number of variables. If you have multiple parts or multiple connections on your GUI application, there is a variable for each, and the variables are sequentially numbered (2B). It creates a new

instance from the AbtShellView class, which is the class that identifies the generic primary part.

For this instance it performs a number of actions. It creates the icons or representations for all the parts in the GUI application and assigns these to a variable (2C).

Next the attribute values for the primary part itself are set. And finally for each of the parts created, identified by their variable, the attributes are initialized, such as the placement and route of the arrow of a connection (2D).

The next section of the .ESF defines the characteristics of each individual part, using the same structure.

Any methods used in this section to create subparts of parts to be created (like the converter) are included in the .ESF.

```
converterPartBuilder: aParentPartBuilder
    "Create the edit time part tree for the AbtStringConverter."

    | anAbtStringConverter |

    anAbtStringConverter := AbtStringConverter newTopLevelPartBuilder.

    |anAbtStringConverter!
```

Public interface

This section (see Figure 59) defines the features that are added to the interface (3A) of the GUI application. This section is optional. Its inclusion depends on the fact that no features other than those of the primary part are part of the interface.

```
+-----+
|
| calculateInterfaceSpec
|     "Create the interface specification for the MEMBER."
|     " MEMBER reinitializeInterfaceSpecFromMethod. "
|
| |AbtInterfaceSpecBuilder new
| | featureBuilderNamed: #featureName put: (AbtSubpartActionSpec newFeatureBuilder
| | attributeSettingNamed: #subpartFeatureName put: #promotedfeature; 3A )
| | attributeSettingNamed: #selector put: #partName;
| | attributeSettingNamed: #subpartName put: 'classNameOfPart');
| | :
| | :
|
+-----+
```

Figure 59. Public Interface Section of an ESF File for a GUI Application

This method returns (|) the interface specification built up with the featureBuilderNamed:put: method. It indicates the name of the feature, the feature that was promoted, the part from which it was promoted, and the class of that part.

At the end of the file, before the escript tag, it also states the version of the product with which the .ESF was created:

```
versionCode
    |16r20000! !
```

The .ESF is a readable file and could therefore be parsed with a REXX program.

C.2 Print File

GUI applications can be printed. The print file can be sent to the printer, and it can be routed to a file. The file has the format as described in "Documentation" in topic 3.1.7.2.

Just like the .ESF file, the print file can be used as the input for a REXX program that analyzes and uses its contents.

C.3 Summary

VisualAge Generator provides two sources of information that can be used to build tools that allow you to analyze or document GUI applications in more detail.

A GUI application can be exported as an external source format (.ESF) file. The file contains the Smalltalk definitions of the methods used to build up the application. The definition can be divided into parts describing the internal structure, the visual layout and the public interface.

The .ESF file can be interpreted using a program such as REXX that parses the output. The same can be done with the print output of a GUI application that is written to a file instead of to a printer.

C.4 What You Should Now Be Able to Do

You should now be able to:

- ☐ Read and understand the readable exported GUI application.
- ☐ Understand the possibilities for supporting your application development effort by using parsing mechanisms for either the .ESF file or the print file.

D.0 Appendix D. Attributes

Each part in VisualAge Generator has a large number of attributes. It takes some time and reading to find your way around the meaning of the different attributes. This appendix will give you a head start by pointing out some of the more important attributes.

HINT
To find a list of all features available in a VisualAge Generator GUI application development environment search on the text "list of features available" in the VisualAge Generator help facility. This will locate the help topic with a list of features with descriptions.

Subtopics

- D.1 Self
- D.2 Object
- D.3 String
- D.4 Items
- D.5 Menu

D.1 Self

Every part in VisualAge Generator has a self attribute. This attribute holds the memory location of the part. This attribute is only used when the attribute you are connecting to requires the memory location of the part. The main examples are:

- self of a variable part. This allows you to act in the variable part just like it is the part to which it is connected.
- items of a container details. Instead of data a container details uses a memory location to get its data from. Each column has an attributeName setting that indicates which part of the data at this memory location should be shown in the column.

The memory location of a part also comes back in other attributes. Data items in a working storage record for instance have two attributes, DATAITEM and DATAITEM data. The first attribute contains the memory location and can be used as a pointer to the data item. You can see this when you tear off this data item. The connection that is created is between the DATAITEM attribute of the working storage record and the self attribute of the variable part that is the tear-off attribute.

The difference also comes back in some actions. For instance, the getFieldAtIndex: action of an occurring data item contains the pointer to the data item in the result attribute of the connection. The associated getValueAtIndex: contains the value of the data item in the result attribute of the connection.

D.2 Object

The object attribute of a part is found in all parts that have a data type associated with them and can contain only one element of data. object contains the value of the part. The part can only have a value that does not violate its data type and data type settings. If the latest value given to the part does, the part shows "*** error **" to indicate an incorrect value. Also the object attribute contains the last value that did not violate the data type and data type settings.

Data type is the most common attribute used to make sure data is shown in a part. The object attribute is also used for the default connection created by the Quick Form facility of VisualAge Generator.

D.3 String

In contrast to object, the string attribute contains a value even if the value provided is incorrect and does not pass the validation. It contains the incorrect value, exactly as typed by the user or provided by the system.

The string value is also updated as a result of any formatting performed by the part, for example, when a currency symbol is placed in front of a digit.

The exact name of this attribute can differ according to the part in which it is used. In an entry field it is string and in a label, labelString.

D.4 Items

Lists of data, like a list box and a container details, have an attribute called items. In general this indicates that the part contains a list of data. If the items attribute is torn off, it results in an ordered collection, a clear list of data.

The type of information the part expects for the items attribute can be different for each list. A container details expects a pointer to the location where it should get its data, whereas a list box wants the real values.

D.5 Menu

A lot of parts have a menu attribute. This attribute allows you to connect a menu to the part. The attribute requires the pointer (self) to the menu to be connected to it. If the part is a window, it shows the menu as part of the window. In all other cases the menu becomes a pop-up menu for the part to which the menu is connected.

E.0 Appendix E. Special Notices

This publication is intended to help application programmers understand and effectively use the GUI application development functions and features of VisualAge Generator. The information in this publication is not intended as the specification of any programming interfaces that are provided by VisualAge Generator Developer. See the PUBLICATIONS section of the IBM Programming Announcement for VisualAge Generator Developer for more information about what publications are considered to be product documentation.

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY 10594 USA.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, Dept. 600A, Mail Drop 1329, Somers, NY 10589 USA.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The information about non-IBM ("vendor") products in this manual has been supplied by the vendor and IBM assumes no responsibility for its accuracy or completeness. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

The following terms are trademarks of the International Business Machines Corporation in the United States and/or other countries:

DB2	DB2/2
IBM	OS/2
VisualAge	VisualGen

The following terms are trademarks of other companies:

C-bus is a trademark of Corollary, Inc.

PC Direct is a trademark of Ziff Communications Company and is used by IBM Corporation under license.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Microsoft, Windows, and the Windows 95 logo are trademarks or registered trademarks of Microsoft Corporation.

Other trademarks are trademarks of their respective companies.

F.0 Appendix F. Related Publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

Subtopics

F.1 International Technical Support Organization Publications

F.2 Redbooks on CD-ROMs

F.3 Other Publications

F.1 International Technical Support Organization Publications

For information on ordering these ITSO publications see "How To Get ITSO Redbooks" in topic BACK_1.

- *Object-Based GUI Application Development with VisualGen, SG24-4233*

Provides a detailed look at object-based GUI application development with sample code.

- *Application Development: 4GL Redbooks, SBOF-7284-00*

Bookshelf of all ITSO redbook publications related to VisualAge Generator.

F.2 Redbooks on CD-ROMs

Redbooks are also available on CD-ROMs. **Order a subscription** and receive updates 2-4 times a year at significant savings.

CD-ROM Title	Subscription Number	Collection Number
System/390 Redbooks Collection	SBOF-7201	SK2T-
Networking and Systems Management Redbooks Collection	SBOF-7370	SK2T-
Transaction Processing and Data Management Redbook	SBOF-7240	SK2T-
AS/400 Redbooks Collection	SBOF-7270	SK2T-
RISC System/6000 Redbooks Collection (HTML, BkMgr)	SBOF-7230	SK2T-
RISC System/6000 Redbooks Collection (PostScript)	SBOF-7205	SK2T-
Application Development Redbooks Collection	SBOF-7290	SK2T-
Personal Systems Redbooks Collection	SBOF-7250	SK2T-

F.3 Other Publications

These publications are also relevant as further information sources:

- ☐ *VisualAge Generator GUI User's Guide and Reference*, SH23-0239-00
- ☐ *Developing VisualAge Generator Client/Server Applications*, SH23-0230-00
- ☐ *Running VisualAge Generator Applications on OS/2, AIX, and Windows*, SH23-0235-00
- ☐ *Generating VisualAge Generator Applications*, SH23-0227-00
- ☐ *Object-Oriented Interface Design, IBM Common User Access Guidelines*, SC34-4399

BACK_1 How To Get ITSO Redbooks

This section explains how both customers and IBM employees can find out about ITSO redbooks, CD-ROMs, workshops, and residencies. A form for ordering books and CD-ROMs is also provided.

This information was current at the time of publication, but is continually subject to change. The latest information may be found at URL **<http://www.redbooks.ibm.com>**.

Subtopics

BACK_1.1 How IBM Employees Can Get ITSO Redbooks

BACK_1.2 How Customers Can Get ITSO Redbooks

BACK_1.3 IBM Redbook Order Form

BACK_1.1 How IBM Employees Can Get ITSO Redbooks

Employees may request ITSO deliverables (redbooks, BookManager BOOKs, and CD-ROMs) and information about redbooks, workshops, and residencies in the following ways:

- ☐ **PUBORDER** -- to order hardcopies in United States
- ☐ **GOPHER link to the Internet** - type **GOPHER.WTSCPOK.ITSO.IBM.COM**
- ☐ **Tools disks**

To get LIST3820s of redbooks, type one of the following commands:

```
TOOLS SENDTO EHONE4 TOOLS2 REDPRINT GET SG24xxxx PACKAGE
TOOLS SENDTO CANVM2 TOOLS REDPRINT GET SG24xxxx PACKAGE (Canadian users only)
```

To get lists of redbooks:

```
TOOLS SENDTO USDIST MKTTOOLS MKTTOOLS GET ITSOCAT TXT
TOOLS SENDTO USDIST MKTTOOLS MKTTOOLS GET LISTSERV PACKAGE
```

To register for information on workshops, residencies, and redbooks:

```
TOOLS SENDTO WTSCPOK TOOLS ZDISK GET ITSOREGI 1996
```

For a list of product area specialists in the ITSO:

```
TOOLS SENDTO WTSCPOK TOOLS ZDISK GET ORGCARD PACKAGE
```

- ☐ **Redbooks Home Page on the World Wide Web**

<http://w3.itso.ibm.com/redbooks>

- ☐ **IBM Direct Publications Catalog on the World Wide Web**

<http://www.elink.ibm.link.ibm.com/pbl/pbl>

IBM employees may obtain LIST3820s of redbooks from this page.

- ☐ **REDBOOKS category on INEWS**
- ☐ **Online** -- send orders to: USIB6FPL at IBMMAIL or DKIBMBSH at IBMMAIL
- ☐ **Internet Listserver**

With an Internet E-mail address, anyone can subscribe to an IBM Announcement Listserver. To initiate the service, send an E-mail note to **announce@webster.ibm.link.ibm.com** with the keyword **subscribe** in the body of the note (leave the subject line blank). A category form and detailed instructions will be sent to you.

BACK_1.2 How Customers Can Get ITSO Redbooks

Customers may request ITSO deliverables (redbooks, BookManager BOOKs, and CD-ROMs) and information about redbooks, workshops, and residencies in the following ways:

- ☐ **Online Orders** (Do not send credit card information over the Internet)
-- send orders to:

In United States:
In Canada:
Outside North America:

IBMMAIL

usib6fpl at ibmmail
caibmbkz at ibmmail
dkibmbsh at ibmmail

Internet

usib6fpl@ibmmail.com
lmannix@vnet.ibm.com
bookshop@dk.ibm.com

- ☐ **Telephone orders**

United States (toll free)
Canada (toll free)

1-800-879-2755
1-800-IBM-4YOU

Outside North America
(+45) 4810-1320 - Danish
(+45) 4810-1420 - Dutch
(+45) 4810-1540 - English
(+45) 4810-1670 - Finnish
(+45) 4810-1220 - French

(long distance charges apply)
(+45) 4810-1020 - German
(+45) 4810-1620 - Italian
(+45) 4810-1270 - Norwegian
(+45) 4810-1120 - Spanish
(+45) 4810-1170 - Swedish

- ☐ **Mail Orders** -- send orders to:

IBM Publications
Publications Customer Support
P.O. Box 29570
Raleigh, NC 27626-0570
USA

IBM Publications
144-4th Avenue, S.W.
Calgary, Alberta T2P 3N5
Canada

IBM Direct Services
Sortemosevej 21
DK-3450 AllerOpen Set
Denmark

- ☐ **Fax** -- send orders to:

United States (toll free)
Canada
Outside North America

1-800-445-9269
1-403-267-4455
(+45) 48 14 2207 (long distance charge)

- ☐ **1-800-IBM-4FAX (United States) or (+1) 415 855 43 29 (Outside USA)** --
ask for:

Index # 4421 Abstracts of new redbooks
Index # 4422 IBM redbooks
Index # 4420 Redbooks for last six months

- ☐ **Direct Services** - send note to **softwareshop@vnet.ibm.com**

- ☐ **On the World Wide Web**

Redbooks Home Page <http://www.redbooks.ibm.com>
IBM Direct Publications Catalog
<http://www.elink.ibm.link.ibm.com/pbl/pbl>

- ☐ **Internet Listserver**

With an Internet E-mail address, anyone can subscribe to an IBM Announcement Listserver. To initiate the service, send an E-mail note to **announce@webster.ibm.link.ibm.com** with the keyword **subscribe** in the body of the note (leave the subject line blank).

BACK_1.3 IBM Redbook Order Form

Please send me the following:

Title	Order Number	Quantity

First name

Last name

Company

Address

City

Postal code

Country

Telephone number

Telefax number

VAT number

Invoice to customer number

Credit card number

Credit card expiration date

Card issued to

Signature

We accept American Express, Diners, Eurocard, Master Card, and Visa. Payment by credit card not available in all countries. Signature mandatory for credit card payment.

DO NOT SEND CREDIT CARD INFORMATION OVER THE INTERNET.

Special Characters

/GENEMBEDDEDGUI 1.1.4

/LINKAGE 1.1.4

A

abstraction 3.2.1.2 3.3.2.2

action 1.0

introduction 1.1.2

sequencing 1.9.1

aggregation 3.2.1.5.1 3.3.2.4

tear-off 3.3.1.2

architecture

application development considerations 3.1.7

business object 3.1.2

controls 3.1.4

design 3.1.7.1

documentation 3.1.7.2

embeddable part 3.1.1

interaction 3.1.6

introduction 3.1

overview 3.1.1

representation 3.1.3

attribute 1.0 1.7.3.2

introduction 1.1.2

B

become primary part 1.1.2

bibliography F.0

bind 1.1.3

breakpoints 1.5.1.1 1.5.1.2

business object

building 3.1.2.1

checklist 3.1.2.2

creating 3.1.2.2

interaction 3.1.6

introduction 3.1.2

representation 3.1.3

views 3.1.3.2

C

characteristics 1.7.5.1

child placement rules 1.4.2.2

class tree 3.2.3.3 3.3.3.1

client/server 2.4.4

color 1.2.1

common

building 3.1.5.1

checklist 3.1.5.2

interaction 3.1.6

introduction 3.1.5

composite parts 1.2.3.1

sizing 1.4.2.2

connections 1.1.2 1.6.2

attribute-to-action 1.6.2.2.4

attribute-to-attribute 1.6.2.2.2

changing 1.6.2.3 1.6.2.3.2

creating 1.6.2.1

event-to-action 1.6.2.2.1

event-to-attribute 1.6.2.2.3

moving 1.6.2.3.4

order 1.6.2.4 1.9.1.1.2

parameters 1.6.2.2.5

performance 2.4.3.2

trigger 1.9.1.1.2

types 1.6.2.2

viewing 1.6.2.3.1

container details

creating 2.1.1.1

editing 2.1.1.4

features 2.1.1.2

introduction 2.1.1

context menu 1.1.2

parts

controls

building 3.1.4.1

checklist 3.1.4.2

creating 3.1.4.2

interaction 3.1.6

introduction 3.1.4

converter 2.3.2.2.1

CUA

standards A.0

VisualAge Generator limitations A.0

D

data display window 1.4.1.1

data element 1.7.1

data entry window 1.4.1.2

- data item 1.7.1.1
 - data type 1.7.1.2
 - defining 1.7.1.3
 - global 1.7.1.1
 - local 1.7.1.1
 - nonnumeric 1.7.1.2.2
 - numeric 1.7.1.2.1
 - toggle 1.9.2.2
- data structure 1.7.3
 - accessing 1.7.3.2
 - attribute 1.7.3.2
 - defining 1.7.3.1
 - multidimensional array 1.7.4.1
 - occurs item 1.7.4
 - tear-off attribute 1.7.3.2
 - using 1.7.3.1
- data type 1.7.1.2
- data type parts 1.2.3.2
- data validation 2.3.2
 - converter 2.3.2.2.1
 - Form Input Checker 2.3.2.2.2
 - masking 2.3.2.1
 - options 2.3.2.2.5
 - VisualAge Generator table 2.3.2.2.3
- database preferences 1.1.3
- debugging 1.5 1.5.2
 - EZERDEBUG 1.5.2.1
 - EZERRUN_DEBUG 1.5.2.1
 - setup 1.5.2.1
 - WALKBACK.LOG 1.5.2.1
- delegation 3.2.1.6
- drag 2.1.2
- drop 2.1.2
- dynamic programming 2.2.2 2.4.1.5
- E**
- embeddable part
 - architecture 3.1.1
 - creating 2.2.1.1
 - defining 2.2.1.1
 - GUI application 1.1.2
 - introduction 2.2.1
 - visible 1.1.2
- encapsulation 3.2.1.3 3.3.2.3
- entry point 3.3.3.3.1
- error handling 2.3
- errors
 - data validation 2.3.2
 - GUI application 2.3.3
 - logging 2.3.4.3
 - messages 2.3.4
 - middleware 2.3.1 2.3.1.1
 - server database errors 2.3.1 2.3.1.2
- event 1.0
- event-driven programming 1.6.1
- events
 - introduction 1.1.2
 - order 1.9.1.1
 - sequencing 1.9.1.2.3
- execution 1.1.5
- eze2db2.bnd 1.1.3
- EZE2RUN
 - PROFILE OFF 1.5.3.1
 - PROFILE ON 1.5.3.1
- EZE2RUN KILL 1.1.5
- EZERDEBUG 1.5.2.1
- EZERRUN_DEBUG 1.5.2.1
- F**
- features 1.0 1.6.2.3.2
 - action 1.1.2
 - attribute 1.1.2
 - changing 1.6.2.3.2
 - connections 1.6.2.2 1.6.2.3.2
 - event 1.1.2
 - settings 1.2.3.4
 - window 1.3.1
- file accessor part 2.1.3
- font 1.2.1
- Form Input Checker 2.3.2.2.2
- free-form surface 1.1.2
 - visual part 1.4.2
 - window 1.4.2
- G**
- generation 1.1.4
 - /GENEMBEDDEDGUI 1.1.4

- /LINKAGE 1.1.4
- performance 2.4.5
- global 1.7.1.1
- GUI application 1.1.2
 - characteristics 1.1.1
 - debugging 1.5
 - definition 1.1.2
 - embedded 1.1.2
 - execution 1.1.5
 - external 1.1.2
 - generation 1.1.4
 - ITF 1.5
 - public interface 1.1.2
 - running 1.1.5
 - saving 1.1.2
 - tabbing order 1.3.3
 - testing 1.1.3 1.5
- GUI application builder
 - parts palette 1.2.1
- GUI application definition window 1.2
- GUI application development
 - characteristics 1.1.1
- GUI builder 1.2
 - parts list 1.3.4
 - tool bar 1.2.1
- GUI programming example
 - ADDRESS 2.2.1.1 2.2.1.3 2.2.1.4.2
 - ADDRVAR 2.2.1.4.2
 - ADDRWIN 2.2.1.1 2.2.1.3 2.2.1.4.1
 - AGGFORM 3.3.2.4
 - AGGWIN 3.3.2.4
 - AIRDATA 2.4.4.2.1
 - AIRTEST 2.4.4.2.1
 - APPENT 3.3.3.3.1
 - BABYBOY 3.1.6.2
 - CDVPAKET 2.1.1.3
 - CNFRMSG 2.1.4
 - CONTST1 1.6.2.1
 - CONTST2 1.6.2.2.5 1.6.3
 - DRAGDROP 2.1.2
 - DYNMENU 2.2.2.3.1
 - DYNPOS 2.2.2.3.2
 - FILEACC 2.1.3
 - FLAG 1.9.2.3
 - FLAGTST 1.9.2.3
 - GUIEMB1 1.1.2 1.4.3
 - GUISHR1 1.7.7.1 1.7.7.2.2
 - GUISHR2 1.7.7.1 1.7.7.2.1
 - GUISHR3 1.7.7.2.1 1.7.7.2.2
 - GUISHR4 1.7.7.2.2
 - GUISHR5 1.7.7.2.2
 - GUITAB1 1.7.5.1
 - GUITAB2 1.7.5.1
 - GUITST1 1.1.2 1.1.3 1.1.4 1.1.5
 - GUITST2 1.1.2 1.1.3
 - GUITST3 1.7.2.1
 - GUITST4 1.7.2.2
 - GUITST5 1.7.3.1 1.7.3.2 1.7.4.1
 - GUITST6 1.7.4.2
 - GUITST7 1.7.6.2
 - GUITST8 1.7.6.3
 - GUITST9 1.7.6.3
 - HOVERHLP 1.3.2
 - ITMTST1 1.7.1.3
 - MEGADATA 2.4.4.2.1
 - MSGFUN 2.3.4.1.2
 - MSGRCD1 1.7.4.1
 - MSGRCD2 1.7.4.1
 - MSGWIN 1.9.2.1 1.9.2.2
 - OFWIN 2.2.2.1 2.2.2.2
 - OPENWIN1 2.2.2.1
 - OPENWIN2 2.2.2.1
 - OPENWIN3 2.2.2.1
 - OPENWIN4 2.2.2.2
 - PRCTST1 1.8.3.1
 - RBBBDDC 3.1.4.2
 - RBBBDDR 3.1.3.2
 - RBBBZDB 3.1.2.2
 - RCDTST1 1.7.2 1.7.2.2
 - RCDTST2 1.7.2
 - RCDTST3 1.7.3.1 1.7.3.2 1.7.4.1
 - RVOCSTEST 3.3.2.5.1
 - SORTESF 3.3.3.1
 - SORTOF 3.3.3.1

- TGRGUI1 2.4.3.2
- TGRGUI2 2.4.3.2
- TGRTAB1 2.4.3.2
- TGRTAB2 2.4.3.2
- VALCONV 2.3.2.2.1
- VALFCHK 2.3.2.2.2
- VALLOG 2.3.4.3
- VALMASK 2.3.2.1
- VALREUSE 2.3.2.2.6
- VALRTST 2.3.2.2.6
- VALTAB 2.3.2.2.3
- VALTST2 2.3.2.2.6
- VGTBLE 1.7.5.1
- VRTCDV 3.3.2.5.3
- VRTCDVT 3.3.2.5.3
- VSLTST1 1.9.1.1.2
- VSLTST2 1.9.1.2.4
- VSLTST3 1.9.1.2.5
- VSLTST4 1.9.2.1
- VSLTST5 1.9.2.2
- VSLTST6 1.9.3.1
- VSLTST7 1.9.3.1
- WINTST1 1.4.3
- H**
- hierarchy 3.2.1.5
- hover help 1.3.2
- I**
- inheritance 3.2.1.5.2 3.3.2.5
- iterator 1.9.3.1
- ITF 1.1.3 1.5
 - bind 1.1.3
 - calling generated server applications 1.1.5
 - database preferences 1.1.3
 - eze2db2.bnd 1.1.3
 - EZERDEBUG 1.5.2.1
 - interrupting 1.5.1.6
 - LUW 1.1.3
 - profile 1.1.3
 - stopping 1.5.1.6
 - test monitor 1.1.3
 - testpoints 1.5.1.1
 - trace entry filter 1.5.1.5
 - trace log 1.5.1.5
- ITF monitor 1.5.1.1
 - bypass 1.5.1.1
 - leap 1.5.1.1
 - return 1.5.1.1
 - run 1.5.1.1
 - stack monitor 1.5.1.1
 - statement monitor 1.5.1.1
 - step 1.5.1.1
 - stop 1.5.1.1
 - watchpoint monitor 1.5.1.1
- L**
- layout 1.4.2.1
- level-77 1.9.1.1.1 2.4.4.2.3
- local 1.7.1.1
- logging 2.3.4.3
- LUW 1.1.3
- M**
- masking
 - data validation 2.3.2.1
 - using 2.3.2.1
- megadata 2.4.4.2.1
- member parts 1.2.3.3
- messages
 - customized 2.3.4.1.2
 - errors 2.3.4
 - logging 2.3.4.3
 - placeholders 2.3.4.1.2
 - table 2.3.4.1.1
 - using 2.3.4.1
 - window 2.3.4.2
- middleware errors 2.3.1
- modes 1.3.1
- modularity 3.2.1.4
- multidimensional array 1.7.4.1
- multiple inheritance 3.3.2.5.2
- N**
- nesting 1.1.2
- O**
- object factory
 - dynamic programming 2.4.1.5.1
 - part 2.2.2.1

- using 2.2.2.1
- object menu 1.1.2
- objects
 - abstraction 3.2.1.2 3.3.2.2
 - aggregation 3.2.1.5.1 3.3.1.2 3.3.2.4
 - building 3.3.2
 - characteristics 3.2.1
 - class tree 3.2.3.3
 - classes 3.3.1.1
 - communication 3.3.2.6
 - delegation 3.2.1.6
 - encapsulation 3.2.1.3 3.3.2.3
 - hierarchy 3.2.1.5
 - implementation 3.3
 - inheritance 3.2.1.5.2 3.3.2.5
 - interaction 3.2.2
 - interface 3.2.3.2
 - modularity 3.2.1.4
 - multiple inheritance 3.3.2.5.2
 - parts 3.3.1.1
 - persistence 3.2.1.7
 - terminology 3.2.3
 - virtual class 3.3.2.5.3
 - VisualAge Generator architecture 3.3.1
- occurs item 1.7.4
 - accessing 1.7.4.2
 - defining 1.7.4.1
 - multidimensional array 1.7.4.1
 - using 1.7.4.1 1.7.4.2
- ordered collection 1.7.6
 - characteristics 1.7.6.1
 - tear-off 1.7.6.3
 - use 1.7.6.2
- P**
- parameters 1.6.2.2.5 1.9.1.2.4
- part
 - position 1.4.2
 - settings 1.1.2
- parts 1.1.2
 - architecture 3.1.1
 - become primary part 1.1.2
 - building 2.2
 - communication 2.2.1.3
 - composite 1.2.3.1 1.4.2.2
 - context menu 1.1.2
 - data type 1.2.3.2
 - delete 1.2.1
 - description 1.2.2
 - embedded 2.2.1
 - member parts 1.2.3.3
 - object factory 2.2.2.1
 - parts list 1.3.4
 - parts palette 1.2.1 1.2.2 2.2.1.2
 - performance 2.4.1.3
 - pop-up menu 1.1.2
 - procedural 1.8.1
 - proportional 1.4.2.1
 - public interface 1.1.2
 - settings 1.2.3.4
 - sharing data 2.2.1.4
 - sizing 1.4.2.2
 - tool bar 1.2.1
 - types 1.2.3
 - using 1.3
 - variable parts 2.2.1.4.2
 - window 1.3.1
- parts list 1.3.4
- parts palette
 - adding parts 2.2.1.2
 - description 1.2.2
 - GUI application builder 1.2.1
 - sticky 1.2.1
- perform request 1.8.3.1 1.9.2.1
- performance 2.4.3.6.2
- performance 2.4
 - array dimensions 2.4.4.2.2
 - client/server 2.4.4
 - connections 2.4.3.2
 - data considerations 2.4.4.2
 - design considerations 2.4.4.1
 - generation options 2.4.5
 - GUI size 2.4.2
 - level-77 2.4.4.2.3
 - load time 2.4.1

- logic 2.4.3
- megadata 2.4.4.2.1
- parts 2.4.1.3
- perform request 2.4.3.6.2
- preloading 2.4.1.2
- tuning 2.4.6
- VisualAge Generator table 2.4.4.2.1
- persistence 3.2.1.7
- pop-up menu 1.1.2
- position 1.4
- primary part
 - become primary part 1.1.2
 - default 1.1.2
- procedural 1.8
- profile
 - database preferences 1.1.3
- promoting 1.7.7.1
- proportional 1.4.2.1
- public interface 1.1.2
- Q**
- quick form 1.7.2.2
- R**
- representation
 - building 3.1.3.1
 - checklist 3.1.3.2
 - interaction 3.1.6
 - introduction 3.1.3
- running GUI applications 1.1.5
 - calling generated server applications 1.1.5
 - ITF 1.1.5
- runtime 1.1.5
 - trace 1.5.3
 - TSCRIPT.LOG 1.5.3.2
- S**
- scrolling 2.1.1.3
- security 3.1.4.1.4 3.3.3.3.2
- server database errors 2.3.1
- settings 1.1.2
 - child placement rules 1.4.2.2
 - data type 1.7.8
 - descriptions 1.2.3.4
 - layout 1.4.2.1
 - parts 1.2.3.4
 - window 1.3.1
- sharing data 2.2.1.4
 - variable parts 2.2.1.4.2
 - VisualAge Generator table 2.2.1.4.3
 - working storage record 2.2.1.4.1
- sizing
 - proportional 1.4.2.1
- sorting 3.3.3.1
- stack monitor 1.5.1.1
- statement monitor 1.5.1.1
- sticky 1.2.1
 - color 1.2.1
 - font 1.2.1
- T**
- tab tags 1.3.3
- tabbing order 1.3.3
- tear-off
 - ordered collection 1.7.6.3
- tear-off attribute 1.7.3.2
- test monitor 1.1.3
- testing 1.1.3 1.5
- testpoints 1.5.1.1
 - breakpoints 1.5.1.2
 - tracepoints 1.5.1.2
 - using 1.5.1.2
 - watchpoints 1.5.1.2
- toggle 1.9.2.2
- tool bar 1.2.1
- trace 1.5.3
- trace entry filter 1.5.1.5
- trace log 1.5.1.5
 - trace entry filter 1.5.1.5
- tracepoints 1.5.1.2
- trigger
 - connections 1.9.1.1.2
 - level-77 1.9.1.1.1
 - order 1.9.1.1.2
- TSCRIPT.LOG 1.5.3.2
- TUI 1.1.1
- tuning 2.4.6
- V**

- validation 2.3.2.2
 - converter 2.3.2.2.1
 - Form Input Checker 2.3.2.2.2
 - options 2.3.2.2.5
 - VisualAge Generator table 2.3.2.2.3
- variable parts 2.2.1.4.2
- virtual class
 - defining 3.3.2.5.3
 - introduction 3.3.2.5.3
- visual part 1.1.2 1.4.2
- VisualAge Generator table 1.7.5 2.2.1.4.3 2.3.2.2.3
 - defining 1.7.5.1
 - performance 2.4.4.2.1
 - using 1.7.5.1

W

- WALKBACK.LOG 1.5.2.1
 - reading 1.5.2.2
- watchpoint monitor 1.5.1.1
- watchpoints 1.5.1.1 1.5.1.2
- widgets 1.1.2
- window 1.1.2 1.3.1
 - data display window 1.4.1.1
 - data entry window 1.4.1.2
 - layout 1.4.2.1
 - modes 1.3.1
 - position 1.4 1.4.2
 - proportional 1.4.2.1
 - tabbing order 1.3.3
 - types 1.4.1
- working storage record 1.7.2
 - accessing 1.7.2.1
 - creating 1.7.2
 - data structure 1.7.3
 - defining 1.7.2
 - multidimensional array 1.7.4.1
 - occurs item 1.7.4
 - quick form 1.7.2.2
 - sharing data 2.2.1.4.1